NPSNET: JANUS-3D Providing three-dimensional displays for a traditional combat model

Walter, Jon Curtis; Warren, Patrick Theron

Monterey, California. Naval Postgraduate School

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED

1b. RESTRICTIVE MARKINGS:

2a. SECURITY CLASSIFICATION AUTHORITY:

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE:

3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S):

5. MONITORING ORGANIZATION REPORT NUMBER(S):

6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School

6b. OFFICE SYMBOL (if applicable): CS

7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School

6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000

7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000

8a. NAME OF FUNDING/SPONSORING ORGANIZATION:

8b. OFFICE SYMBOL (if applicable):

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER:

8c. ADDRESS (City, State, and ZIP Code):

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)
NPSNET: JANUS-3D, Providing Three-Dimensional Displays for a Traditional Combat Model (U).

12. PERSONAL AUTHOR(S)
Walter, Jon C. and Warren, Patrick T.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 08/90 TO 09/92	1992, September 24	104

16. SUPPLEMENTARY NOTATION: The views expressed in this thesis are those of the author and do not reflect the offici policy or position of the Department of Defense or the United States Government.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Virtual World, Three-Dimensional, Combat Modeling, C Programmin Language, Terrain Generation, Real Time, Networking, JANUS, NPSNET

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This work involves integrating the Army's existing combat modeling tool, JANUS, with the real-time three dimensional graphics display offered by NPSNET. The development of a portable software package that can creat a three-dimensional virtual world from any existing JANUS terrain database is explained. In addition, a scripting too capable of rendering JANUS scenarios previously executed in the traditional two-dimensional model is discussed This replay capability allows the gamer/analyst the ability to watch the three-dimensional battle unfold from an position on the battlefield. Lastly, the implementation of a real-time, networked link from the two-dimensiona JANUS model to NPSNET is detailed. This link involves an Ethernet connection from a Sun workstation, whic houses the two-dimensional model, to a Silicon Graphics workstation used for rendering the real-time three dimensional simulation. The methodology used and techniques developed are fully portable to any workstation wit X-windows capability and any graphics workstation equipped with the GL libraries.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
[X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr. Michael J. Zyda

22b. TELEPHONE (Include Area Code): (408) 646- 2174

22c. OFFICE SYMBOL: CS/37

Approved for public release; distribution is unlimited

NPSNET: JANUS-3D
Providing Three-Dimensional Displays for a Traditional Combat Model
by

Jon Curtis Walter
Captain, United States Army
B. S. Engineering, United States Military Academy, 1983

and

Patrick Theron Warren
Captain, United States Army
B. S. Engineering, United States Military Academy, 1982

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

September 1992

Robert B. McGhee, Chairman,
Department of Computer Science

ABSTRACT

This work involves integrating the Army's existing combat modeling tool, JANUS, with the real-time three-dimensional graphics display offered by NPSNET. The development of a portable software package that can create a three-dimensional virtual world from any existing JANUS terrain database is explained. In addition, a scripting tool capable of rendering JANUS scenarios previously executed in the traditional two-dimensional model is discussed. This replay capability allows the gamer/analyst the ability to watch the three-dimensional battle unfold from any position on the battlefield. Lastly, the implementation of a real-time, networked link from the two-dimensional JANUS model to NPSNET is detailed. This link involves an Ethernet connection from a Sun workstation, which houses the two-dimensional model, to a Silicon Graphics workstation used for rendering the real-time three-dimensional simulation. The methodology used and techniques developed are fully portable to any workstation with X-windows capability and any graphics workstation equipped with the GL libraries.

CPT Walter concentrated his efforts in the areas of translations of JANUS(A) files to NPSNET file format and the networking of both JANUS(X) and NPSNET. CPT Warren's primary focus was on generating three-dimensional terrain and the writing of JANUS(A) scripts that NPSNET can read. Both CPT Walter and CPT Warren worked together to create the tree and city canopies and other cultural features.

C.1

TABLE OF CONTENTS

I. INTRODUCTION 1
A. BACKGROUND 1
B. MOTIVATION 1
C. OBJECTIVES 2
D. CHAPTER SUMMARY 3
II. JANUS(A): OVERVIEW 4
A. BACKGROUND 4
B. DESCRIPTION 4
C. HARDWARE 5
D. PROGRAM SUMMARY 5
E. DATA FILE MANAGEMENT AND CONTENT 6
1. JANUS(A) Initialization Files 7
a. Terrain Files 7
b. System Files 8
c. Force Files 9
d. Deployment Files 9
2. JANUS(A) Post-Processor Files 9
III. TWO-DIMENSIONAL TERRAIN AND SCRIPTING 11
A. SGI FORMATTED JANUS(A) FILES 11
1. Initialization Files 11
a. Terrain Files 11
b. System Files 12
c. Force Files 13
d. Deployment Files 13
2. SGI Formatted Post-processor Files 13
B. REPLAY 13
1. Two-Dimensional Map Generation 14
2. Two-Dimensional Scripting Capability 16
IV. NPSNET: INTEGRATION 19
A. BACKGROUND 19
B. HARDWARE 19
C. DESCRIPTION 19
D. NPSNET VEHICLE MOVEMENT AND SIMULATION 20
1. Vehicle Models 20
2. Vehicle Data Structure 20
3. Vehicle Activity in NPSNET 22
E. GENERATION OF NPSNET ELEVATION FILES 23
1. JANUS(A): Coordinate System. 23
2. NPSNET: Coordinate System 24
F. GENERATING MESH TERRAIN 25

1. Creating the Binary Elevation file. 25
2. Creating Mesh Terrain 25
G. GENERATION OF POLYGONIZED TERRAIN SKIN 26
1. Structure of Polygonized Terrain 26
a. Description of Grid Node 26
b. Polygonized Terrain Resolution 27
c. Description of a Terrain Quadtree 28
2. Generating Terrain Polygons 30
H. GENERATION OF NPSNET ROADS AND RIVERS 31
I. GENERATION OF NPSNET CITIES AND TREES 32
1. A Three-Dimensional Interpretation of Two-Dimensional Objects 32
2. Generation of City and Tree Canopies 33
a. Description of a Canopy 34
b. Determining the Canopy Height 35
c. Construction of Canopy Sides 35
3. Placement of Individual Trees and Buildings 38
a. Tree Placement 39
b. Building Placement. 40
c. Equations for Determining Line-Polygon Intersections 41
V. NPSNET: GENERATION OF A THREE-DIMENSIONAL SCRIPT 43
A. THREE-DIMENSIONAL INFERENCE PROBLEM 43
B. APPROACH 44
C. CREATING THE NPSNET SCRIPT FILE 44
D. INFERRED INFORMATION REQUIREMENTS 46
E. BASIC CALCULATIONS 46
F. THE HEURISTIC MODEL 48
1. Defensive Heuristic 48
2. Offensive Heuristic 49
3. Speed Smoothing Heuristic 49
4. Vehicle Orientation Heuristic 50
5. Weapon Orientation Heuristic 50
G. CALCULATION OF THE HEURISTICS IN CLIPS 50
VI. NETWORKING AND REAL TIME SIMULATION 53
A. JANUS(A) FOR THE UNIX OPERATING SYSTEM 53
1. Background 53
2. Contrasts Between the UNIX and VAX Versions of JANUS(A) 53
3. JANUS(X) Networking Capabilities 54
4. Network Message Format 56
a. JANUS(X) Message Format 56
b. NPSNET Message Format 58
B. NPSNET MESSAGES FROM JANUS 59
1. General Message Translation to NPSNET Format 59
a. Segment Number Conversion 59

b. Grid Coordinate Translation....59
c. Vehicle Model Identification....60
d. Elimination of Multiple Update Commands....61
2. NPSNET Initialization....62
3. Vehicle Movement Updates....64
4. Vehicle Shots....65
5. Artillery....65
6. Vehicle Destruction....66
VII. CONCLUSIONS AND RECOMMENDATIONS....67
A. CONCLUSIONS....67
1. Results....67
2. Applications....67
B. RECOMMENDATIONS....68
APPENDIX A: FILE TRANSLATION TECHNIQUES....69
APPENDIX B: USER'S GUIDE....72
A. INTRODUCTION....72
B. VAX TO UNIX FILE TRANSLATION....72
1. Translation of Terrain File....73
2. Recompilation....73
3. Translation of the System File....74
4. Translation of the Force File....74
5. Translation of the Deployment File....75
6. Translation of Post-processor Files....75
C. TERRAIN GENERATION....75
D. SCRIPT GENERATION....80
E. Operation of the Two-Dimensional REPLAY Program....81
1. Initialization and Start-Up Procedures....81
2. Program Manipulation....82
F. Operation of NPSNET: JANUS-3D....84
1. NPSNET: JANUS-3D Compilation....84
2. Operation of NPSNET: JANUS-3D in Script Mode....85
3. Operation of NPSNET: JANUS-3D in Real-Time Mode....87
LIST OF REFERENCES....91
INITIAL DISTRIBUTION LIST....93

LIST OF FIGURES

Figure 1, Fulda Gap Being Rendered by Grey-Scaling Algorithm. 15
Figure 2, Fulda Gap with Optional Terrain Features. 16
Figure 3, Fulda Gap with Roads and Scripted Battle. 18
Figure 4, Vehicle Information Data Structure 21
Figure 5, Description of JANUS Map Resolution 24
Figure 6, NPSNET: Coordinate System 25
Figure 7, Tri-Mesh Terrain Traversal Pattern 26
Figure 8, Basic Terrain Node 27
Figure 9, Quadtree: Terrain Illustration 29
Figure 10, Quadtree: Data Structure Illustration 29
Figure 11, Algorithm to Adjust Map Size According to Quadnode Requirements 30
Figure 12, Quadnode File Representation 31
Figure 13, Drawing of a Tri-Mesh Canopy 34
Figure 14, Order of Points for T-Mesh Canopy Top 36
Figure 15, Algorithm to Draw the Canopy T-Mesh. 37
Figure 16, Cultural Object Description 38
Figure 17, Polygon Intersection Examples 39
Figure 18, Tree Placement Algorithm 40
Figure 19, Illustration of Line Segment Intersections 42
Figure 20, The Traditional JANUS(A) Hardware Setup. 54
Figure 21, Networking Capabilities of JANUS(X) 55
Figure 22, JANUS(X) Network Message Example 58
Figure 23, Vehicle Type Identification 61
Figure 24, Sample Network Message Code 62

Figure 25, NPSNET Initialization Message Function 64
Figure 26, Big-Endian vs. Little-Endian Architectures. 69
Figure 27, VAX Architecture, Four-Byte, Floating Point Format 70
Figure 28, Example Code for VAX to SGI Float Conversion 71
Figure 29, List of Directories Requiring Recompilation 74
Figure 30, Terrain Data Directories 76
Figure 31, REPLAY Main Menu and Terrain Feature Sub-Menu 82
Figure 32, Replay Events Menu and Clock Speed Sub-Menu 83
Figure 33, NPSNET Main Menu and Script Sub-Menu 86
Figure 34, Makejanwin Shell Script 87
Figure 35, Example of Aliased Commands for JANUS(X) Initialization 88
Figure 36, Aliased Command to Run JANUS on Gravy1 Terminal 88
Figure 37, Example of the RED SIDE JANUS(X) Screen 90

ACKNOWLEDGMENT

As with all projects of this magnitude, there are many people who contributed to its success.

First, and foremost, David R. Pratt must be praised for initially envisioning the project. We would like to thank him for the continuing inspiration he provided throughout our work. For his readily available and unceasing advise, expertise, and guidance, we are extremely grateful.

Sincere gratitude goes to Dr. Michael J. Zyda for his timely critiques of our work. We also thank him for creating a "graphics track", and providing us the necessary tools and skills to complete our research.

Special thanks is owed to Rosalie Johnson for her invaluable advise and expertise concerning the many software compatibility problems we encountered. Without her help we would not have finished.

This project would never have gotten off the ground without the first-hand expertise provided by Mr. Al Kellner of TRAC-WSMR. We appreciate his untiring and detailed description of the inner workings of JANUS(A), during our "All Night Cram Session". But most of all, for his inspiring challenge -- "it can't be done!".

Also, Mr. Jim Guyton, of the Rand Corporation, is thanked for his extreme patience when pestered by our numerous phone calls concerning the UNIX version of JANUS. His advice was invaluable.

A very warm and special thank you is duly owed to Captain David A. Nash. On many occasions, too numerous to count, Dave provided insightful critiques, invaluable debugging, and competent expertise on programming languages and hardware. This, coupled with his genuine concern and fellowship, should have earned his name a place on the front of this thesis.

Lastly, we thank our families, who demonstrated incredible patience, understanding, and support, during our many long hours of work. We could not have done it without you!

I. INTRODUCTION

A. BACKGROUND

In a time of shrinking budgets and growing demands, the key challenge for military trainers is to find the most effective and efficient use of technology to supplement traditional instruction. Motivated by this need, the Graphics and Video Laboratory of the Department of Computer Science at the Naval Postgraduate School is currently conducting research on building low-cost, three-dimensional visual simulators on commercially available graphics workstations. The goal of these simulators is the generation of three-dimensional views of terrain, cultural features and vehicles using readily available databases. Current visual simulation efforts are on the NPSNET system, a networked visual simulator that utilizes SIMNET databases and SIMNET networking formats [ZYDA 91] [ZYDA 91a] [ZYDA 92] [PRAT 92]. One of the most recent research developments underway includes the integration of NPSNET with the U. S. Army's two-dimensional combat model, JANUS(A).

B. MOTIVATION

In 1991, the U. S. Army Training and Doctrine Command selected JANUS(A) as the simulation software standard for training at company and platoon levels, using the battalion commander as the senior trainer. It is also used in a seminar role by the Command and General Staff College to train new battalion and brigade commanders on the principles of synchronized combined arms operations [KANA 91]. JANUS(A) is fielded throughout the world and, because of its ability to accurately model complex combat scenarios, is widely used by trainers and analysts in numerous applications which include combat training, studies of combat operations, combat development, testing of new equipment, and research and development.

Despite its great success and huge popularity as a combat development and testing tool, JANUS(A) analysts and developers realized the need for a three-dimensional view of the battlefield to validate their results. An example of this is the new system testing of the

M1A2 main battle tank at Fort Hunter Liggett, California. As part of this testing, the U. S. Army TRADOC Analysis Command, Monterey, used JANUS(A) as a tool to model the operational testing of the M1A2 tank, before it was actually tested on the ground. A three-dimensional capability would help validate this data by verifying events, such as tank positions, direct fire engagements, and line of sights between weapon systems.

Also, because of past virtual reality successes, such as SIMNET, users knew that a three-dimensional view, in support of JANUS(A) exercises, would greatly enhance training simulations. Almost as good as being on the actual terrain, a three-dimensional world would give the gamer/analyst the ability to watch the battle unfold from any position on the battlefield. In addition to standard scenarios, this ability could be used, in conjunction with JANUS(A), for three-dimensional 'after-action reviews' at Combat Training Centers, such as the National Training Center, Fort Irwin, California. Likewise, previously fought battles, such as the famous 'Battle of 73 Easting', could be reenacted on JANUS(A), and refought in the three-dimensional simulation, allowing for several 'what-if' conditions.

C. OBJECTIVES

The primary objective of this work is to produce a working three-dimensional virtual world that is fully integrated with JANUS(A). To accomplish this objective, the goals were threefold: (1) create a three-dimensional virtual world from any existing JANUS(A) terrain database, (2) write a scripting tool capable of rendering JANUS(A) scenarios on this three-dimensional terrain that were previously executed in the traditional two-dimensional model, and (3) implement a real-time, network link from the two-dimensional JANUS(A) model to NPSNET.

The methodology consisted of first developing a two-dimensional model to validate both the terrain and the script data derived from JANUS(A), followed by the integration of JANUS(A) with a fully functioning three-dimensional viewing capability.

D. CHAPTER SUMMARY

Chapter II gives an overview of the JANUS(A) system, including the background of its development, a brief description of the model, and the hardware necessary to run it. In addition, this chapter discusses the many different programs that are included in JANUS(A), followed by a description of the different data files, and their relationship with these programs.

The different hardware systems caused a data file compatibility problem, thus JANUS(A) files were converted to a format compatible with NPSNET. Chapter III describes the file management system developed for these converted files. It also discusses the two-dimensional replay model which was developed to validate both the converted terrain model and the accuracy of the conversion utilities.

Chapter IV provides an overview of NPSNET and the methods used to integrate NPSNET with the JANUS(A) combat model. This includes a discussion of the creation of three-dimensional terrain, cities, and trees.

One problem, when integrating the two-dimensional model with NPSNET, was that JANUS(A) does not produce all of the information necessary to define what a vehicle is actually doing every moment of the game. Chapter V describes, in detail, the expert system used to convert two-dimensional post-processor scripts to a three-dimensional script.

Chapter VI describes the implementation of a real-time, network link from the recently developed Sun-based version of JANUS(A) to NPSNET.

Two appendices are provided. Appendix A describes, in detail, the method used to translate VAX formatted files to IEEE format. Appendix B is a users guide for the JANUS-3D program.

II. JANUS(A): OVERVIEW

A. BACKGROUND

JANUS, named for the ancient Roman god who guards portals, is an interactive, computer based, war-gaming simulation of combat operations conducted at the brigade and lower level in the United States Army [JANU 86]. The original JANUS simulation began in the late 1970's at the Lawrence Livermore National Laboratory (LLNL) to model nuclear effects, and gained considerable reputation for its innovative use of graphical user interfaces [KANA 91]. The U. S. Army TRADOC Analysis Command, White Sands Missile Range, New Mexico (TRAC-WSMR), acquired this prototype from LLNL as a result of the JANUS Acquisition and Development Project, directed by the U. S. Army Training and Doctrine Command (TRADOC) in 1980 [JANU 86]. In 1983, TRAC-WSMR adopted JANUS and further developed it as a high resolution simulation to support analysis for Army combat developments.

The original version, developed at LLNL is known as JANUS(L), while the model developed by TRAC-WSMR is known as JANUS(T). Subsequent to their development, both of these models gained in popularity and employment by a varied number of users, which led to a wide proliferation of different versions of both models. The JANUS(Army) Program began in 1989 to solve the standardization problem and to field a single version, JANUS(A), for all Army users. Today, JANUS(A) is developed, maintained, and distributed by TRAC-WSMR, and is fielded throughout the world as a tool for both trainers and analysts in research and development, testing, and combat development.

B. DESCRIPTION

JANUS(A) is a "two-sided, interactive, closed, stochastic, ground combat simulation" [JANU 91]. It is termed 'two-sided' because it allows the simulation of two opposing forces. These two forces, the Blue force and the Red force, are simultaneously directed and controlled on separate monitors by two different sets of players. Each monitor displays only the vehicles pertaining to its side, plus the opposing vehicles which are directly observed

by its vehicles. Therefore, the model is classified 'closed' because the friendly force player does not know the complete disposition of the opposing forces. The model is 'interactive' because each player monitors, directs, reacts to, and redirects all key actions of the simulated units under his control. Once a scenario is started, certain events in the game, such as direct fires and artillery impacts, are 'stochastically' modelled, which means that they act according to the laws of probability, and thus are different for every scenario run. The principal modeling focus in JANUS(A) is on military systems that participate in maneuver and artillery operations on land, thus the term 'ground combat simulation'.

C. HARDWARE

JANUS(A) currently runs on any Digital Equipment Corporation (DEC) VAX family of computer systems utilizing the standard VMS operating system. A minimal hardware configuration (MV3100E with 12MB memory and four Tektronix 4225 workstations) costs about $85,000 [JANU 91]. In August 1991, the Army directed that JANUS(A) be fielded on an "open system". Since then, it has been successfully demonstrated on UNIX based X-workstations, and has been benchmarked as an open system for August 1992 [KANA 91].

D. PROGRAM SUMMARY

JANUS(A) is composed entirely of Army-developed algorithms and data to model combat processes. The multitude of programs which belong to JANUS(A) consist of approximately 200,000 lines of code written entirely in VAX-11 FORTRAN, a structured Digital Equipment Corporation (DEC) extension of ANSI standard FORTRAN-77 [JANU 91]. In addition to these combat simulation programs, JANUS(A) also has eleven utility programs to facilitate the creation, running, and after-action analysis of a specific scenario. These programs are outlined in Table 1.

TABLE 1: JANUS(A) PROGRAMS AND SUMMARY DESCRIPTION

Program	Description
TRNFLTR	Creates terrain files suitable for use by JANUS(A), using any TRAC-WSMR DMA terrain database as input
TED	Allows interactive editing of JANUS(A) terrain files, and the building of JANUS(A) display map data files
SYMBOLS	Allows interactive design of graphical symbols to represent each type of system being modeled
CSDATA	Interactively maintains the large permanent JANUS(A) system performance database
FORCE	Allows interactive specification of the number of each type of system comprising each force in a particular scenario
MERGE	Allows a scenario to be built from the Blue Forces of one scenario, and the Red Forces of another scenario
VFYSCEN	Reviews all system and force data for a particular scenario, and reports errors or inconsistencies
GRAFVFY	Graphically displays each weapon/system capability and sensor capability against opposing force systems
INITSCEN	Initializes an existing scenario for use as a new scenario, or creates a scenario from scratch
POSTP	Provides a history and reports of nearly every event which occur during a JANUS(A) run
JAAWS	Provides a quick graphical review of force movement, detections, and attrition which occurred during a JANUS run

E. DATA FILE MANAGEMENT AND CONTENT

Every program included in JANUS(A) is dependent on the reading and/or writing of many data files. Therefore, fully integrating JANUS(A) with the three-dimensional capabilities of NPSNET involves a complete understanding of the content of the various files, and the read/write relationship between the data files and the different programs.

Below is a description of the organization and content of the JANUS data files used to set up the system (initialization files), and the JANUS post-processor files used to run the scripted scenarios.

1. JANUS(A) Initialization Files

In the combat model JANUS, the initial state of each scenario is recorded in the files *force###.dat*, *system###.dat*, *dploy###.dat* and *terain###.dat*. For the first three files, '###' is a three digit number from 001-999 which refers to a specific scenario. In the terrain file, '###' is a three digit number from 100-999 which corresponds to the actual digitized terrain file used in the model. These files include information concerning the two-dimensional map, the types and initial locations of vehicles and other data pertaining to the characteristics of the vehicles. The JANUS(A) data files are all stored in VAX binary format. The following is a brief description of each of the JANUS(A) file formats.

a. Terrain Files

The file *terain###.dat*, generated by the programs 'TED' and 'TRNFLTR', contains all of the physical terrain information that JANUS requires to represent the two-dimensional map. The terrain resolution is variable, with the standards being 25, 50, 100, and 200 meter terrain grids. The header information in each of the binary terrain files consists of the following specific information:

(1) The X and Y UTM coordinates of the lower left corner of the map.

(2) The width and height of the map in kilometers.

(3) The number of grid cells in the X and Y direction.

(4) Movement factors for urban and vegetated areas.

(5) Vegetation and urban heights and probability of line of sights.

In addition, each terrain file contains an array of river nodes followed by an array of road nodes. Each node in the array contains the X and Y coordinate of the node followed by a boolean. If the boolean is positive the next node belongs to the present node.

If the boolean is zero, the next node belongs to a new road or river. And, if the boolean is negative, it signifies the last node in the array.

The largest portion of the terrain file is the data stored for each terrain grid cell, which includes:

(1) Elevation of the grid cell
(2) Density information
(3) City or tree present flag, and its respective height
(4) Trafficability and terrain roughness information
(5) River present flag
(6) Obstacle present flag
(7) Flags representing smoke, HE, chemical, radiation, or fire present

b. System Files

JANUS(A) has a very large database which contains the performance data for every possible system, weapon, sensor, mine, barrier, and other entities that can be simulated by the model. Likewise, new combat systems or proposed systems can be added to the database. Because this database is so large, there are limits on the number of different systems which can be modeled at one time. Thus, when a new scenario is being built, the system types of the units being represented are copied from this database to a smaller file to support the scenario. This smaller file is the *system###.dat* file, and is created by running the utility program 'FORCE'.

The data comprising each system type is enormous, but includes attributes/ characteristics such as:

(1) Names and documentation for each system
(2) Road speed
(3) List of direct fire weapons and sensors attached
(4) Vulnerability to artillery, direct fire, and mines
(5) Probability of hit vs. range by posture, aspect, and motion
(6) Probability of kill given a hit vs. range by posture and aspect

[JANU 91].

c. Force Files

In addition to writing the new scenario database, *system###.dat*, the 'FORCE' program also creates the data file *force###.dat*. This file, which is unique for each different type of scenario being modelled, contains the specific force structures of the friendly and opposing units. In other words, the units are divided by side/monitor and further subdivided by task forces. Each side in a JANUS scenario can accommodate 600 separate vehicles for a total of 1200 vehicles modelled. The specific data elements in the force file are:

(1) System type, unit side, and task force.

(2) Number of elements per individual system.

(3) Number of mines, smoke grenades, mine clearing equipment, etc.

d. Deployment Files

The data file *dploy###.dat* contains all of the initialization criteria for each unit which has been designated in the *force###.dat* file. The program 'INITSCEN' creates this deployment file, and must be run each time a scenario is changed from an existing scenario or created from scratch. The specific items contained in the deployment file are:

(1) Unit positions and movement routes.

(2) Mount status

(3) Planned direct fire and artillery missions

(4) Barrier locations.

2. JANUS(A) Post-Processor Files

All significant events which occur during a JANUS(A) run are recorded in disk files, called post-processor files, which can be printed in a human readable format to the screen. The program 'POSTP' is available to the JANUS(A) user to produce these historical reports, which are used to analyze certain aspects of the battle. Since one of the goals was to produce a three-dimensional playback capability of a JANUS(A) scenario, these files were significantly important to understand. There are nine total post-processor

files, but only five of them, *ppmove###.dat*, *ppfirs###.dat*, *pparty###.dat*, *ppkils###.dat*, and *ppmins###.dat*, were actually needed to create a script capability. In all five files, the number '###' refers to the specific scenario. Like the initialization files, these files are stored in VAX binary format. The following is a brief description of each:

TABLE 2: JANUS(A) POST-PROCESSOR FILES AND DESCRIPTION

PP File	Description
PPMOVE	Sequential list, by game time, of the location of every unit, by vehicle number and side.
PPFIRS	Sequential list, by game time, of the location and speed of the firer and the target.
PPARTY	Sequential list, by game time, of the location of the firer, type of volley, number of rounds fired, and location of impact.
PPKILS	Sequential list, by game time, of the location of the firer and the victim and the kill type.
PPMINS	A record of all minefields, fascams, and gems activated during the run. Includes the location, type, size, number of mines, etc.

III. TWO-DIMENSIONAL TERRAIN AND SCRIPTING

The first step in developing a realistic three-dimensional virtual world and a representative scripting capability for previously run scenarios was to build a two-dimensional prototype. This system, run entirely on an SGI/IRIS, was extremely effective in testing the validity of the converted terrain model and the accuracy of the conversion utilities developed to transform the JANUS(A) initialization and post-processor files into a format usable by NPSNET. The different hardware systems caused a significant compatibility problem. Appendix A discusses the methods used to convert VAX formatted files to a format compatible with the Silicon Graphics Computers. This chapter describes the file management system developed for the converted files. It also discusses the two-dimensional replay model which was developed.

A. SGI FORMATTED JANUS(A) FILES

This section briefly describes the format and content of the converted JANUS(A) initialization and post-processor files. For simplicity in reading and validating the data, all of the VAX to SGI converted data are stored as text files.

1. Initialization Files

As stated in Chapter II, the initial state of each scenario is recorded in the files *force###.dat*, *system###.dat*, *dploy###.dat* and *terain###.dat*. These files include information concerning the two-dimensional map, the types and initial locations of vehicles, and other data pertaining to the characteristics of the vehicles.

a. Terrain Files

The conversion program for the terrain files is called *readtrrn.c*. It takes, as input, the three digit number '###' signifying the digitized map used in the model. The corresponding JANUS(A) terrain file, *terain###.dat*, is first decoded and then written into three different text files corresponding to the grid cell data, the river array data, and the road array data. The grid cell data is stored in the file *###.ele*, and begins with a header

containing the lower left X and Y coordinates, the number of grid cells, and the map resolution. The remainder of the grid cell data is formatted as shown in the table below.

TABLE 3: CONVERTED GRID CELL DATA FILE FORMAT

X Cell	Y Cell	Elevation	Density	City/Tree	...	HE
0	0	324.0	5	0	...	0
1	0	332.0	7	1	...	0
...						
120	120	333.0	3	0	...	0

The river array and road array data are contained in *###.riv* and *###.road* respectively. They each contain a line entry for the X coordinate of a node, followed by the Y coordinate, and lastly a boolean representing (1) the continuation of a river/road, (2) the end of a river/road, or (3) the end of all rivers/roads.

b. System Files

The conversion program for the system files is called *readsysfiles.c*. It takes, as input, the three digit number '###' signifying the scenario number being used in the model. As mentioned in Chapter II, this system file is enormous and contains almost every conceivable bit of performance data available for any weapon system. Since the system only simulates what JANUS(A) has already modeled in a scenario, most of the information in this system file is not needed. Thus, after decoding *system###.dat*, information is written to only three different text files. The first, *sys_force.dat*, contains all of the information needed to match the proper icon symbol to the appropriate system. The next, *name.dat*, is a table matching the different system numbers to the actual assigned JANUS system name. And the last, *cloud.dat*, contains the cloud ceiling, wind direction and speed, and the cloud coefficients.

c. Force Files

Once the system file has been converted, the conversion program for the force file, *readforcefile.c* can be run. Like the others, it accepts the three digit number '###' signifying the scenario number being used in the model. For the same reasons as the system file, much of the data written in *force###.dat* is not needed for the simulation. After decoding the force file, information is written to three different text files. The first, *sysnames.dat*, matches a unique vehicle to its specific system number and text name. The second, *force.data*, matches a unique vehicle to its side, vehicle number, vehicle type, task force, and number of subordinate elements. The last file, *icontable.dat*, matches a unique vehicle to its appropriate two-dimensional JANUS(A) icon symbol.

d. Deployment Files

The conversion program for the deployment file is *readdeploy.c*. It also accepts the three digit number '###' signifying the scenario number being used in the model. After decoding the deployment file, *dploy###.dat*, one text file is written. This file, *initunit.dat*, simply contains the initial positions and orientations for every vehicle in the scenario.

2. SGI Formatted Post-processor Files

The conversion program for the post-processor files is *readppfiles.c*. It accepts the three digit number, '###', matching the scenario which has previously been run and saved. It sequentially decodes all five post-processor files, then writes all of the data from each into their respective text files. For example, the JANUS(A) post-processor file, *ppmove###.dat*, would be transformed from VAX to SGI format, then written into the text file *ppmove.dat*.

B. REPLAY

The next step, after (1) converting and formatting the JANUS(A) data files from VAX to IEEE and (2) arranging them in the new file management system, was to test the display

of the two-dimensional terrain, vegetation, and cultural features on the prototype system. In addition to the map display, the validity of the initialization and post-processor files were tested by creating a two-dimensional scripting capability. This section discusses this two-dimensional terrain and scripting program, 'REPLAY'.

1. Two-Dimensional Map Generation

All of the information needed for the creation of the two-dimensional map, except for the roads and rivers, is found in the converted file *###.ele* (see table 3). The most important part of this file is the elevation data that is provided for each grid cell. As mentioned in Chapter II , there are several different resolutions available for any particular piece of terrain. The elevations given for a specific grid cell correspond directly to the resolution of the map. For example, if a map had a resolution of 100 meters, each elevation would correspond to a 100 x 100 meter section of ground. To accurately portray this terrain, the elevations are represented by ring each one a brighter shade of grey from the minimum to the maximum elevation. Since there are 255 different available shades of grey, the proper grey-scale is calculated by using the ratio shown in Equation 3.1 [PRAT 90].

$$COLOR = (ELEVATION_{MIN} + 255 \times (\frac{ELEVATION - ELEVATION_{MIN}}{ELEVATION_{MAX} - ELEVATION_{MIN}})) \qquad \text{(Eq 3.1)}$$

Using this equation resulted in an esthetic view and accurate representation of the two-dimensional map. An example of a 100 meter resolution, 12 x 12 kilometer section of terrain being rendered by the program, 'REPLAY', is shown in Figure 1.

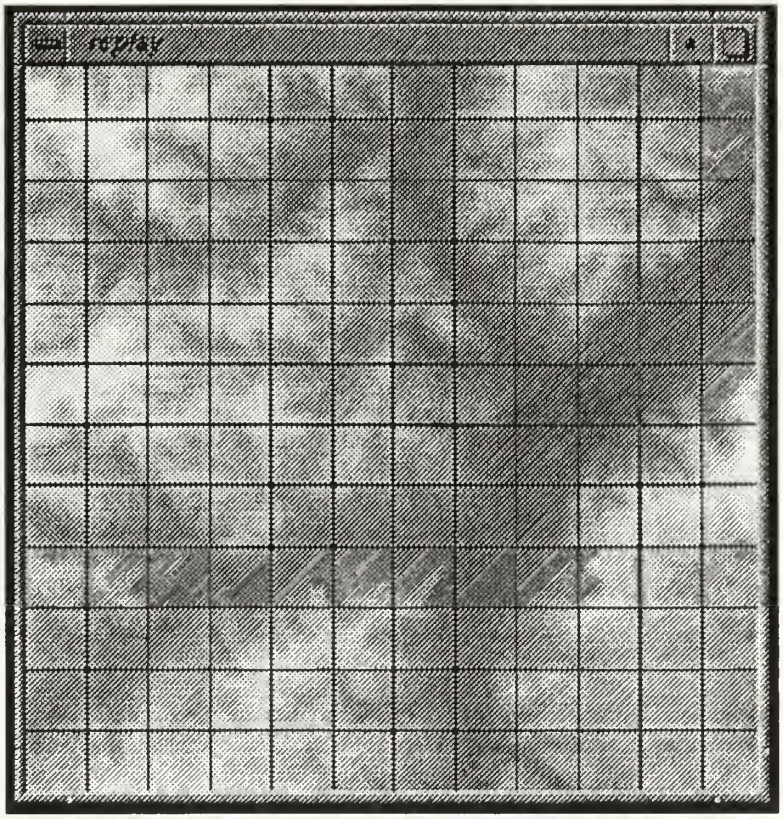

Figure 1: Fulda Gap Being Rendered by Grey-Scaling Algorithm.

As discussed above, the *###.ele* file contains much more than just elevations. All of these pieces of information are displayed to the screen in a similar manner as the grey-scale map. The cities are displayed in shades of dark grey, and the vegetation is displayed in shades of green, each corresponding to their appropriate density values. Yet, unlike the grey-scale map, which is permanently displayed on the screen, the remainder of the terrain and cultural features is displayed by a pop-up menu. These features are then be rendered or hidden as needed.

The road and river data, contained in their own separate files, is simply a collection of connected nodes. These features are also rendered through a pop-up menu, which simply connects the nodes with black lines for roads and blue lines for rivers. An example of cities, vegetation, roads, and rivers is shown in Figure 2. It is being displayed on the same grey-scaled map featured in Figure 1.

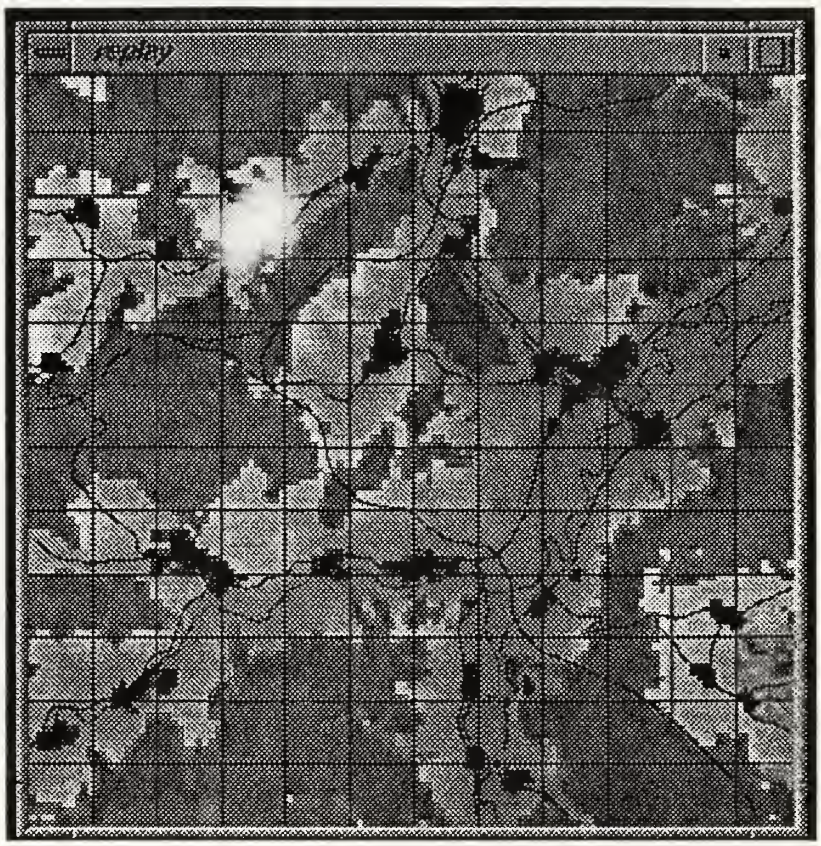

Figure 2: Fulda Gap with Optional Terrain Features.

2. Two-Dimensional Scripting Capability

With the terrain file verified, the next step was to produce a representative two-dimensional script of a scenario which had been previously run on the traditional JANUS(A) model. The appropriate vehicles are successfully matched to their side (red or blue) and to their particular icon symbol by using the converted force and system files. Then, the converted deployment file is used to properly render each vehicle, according to its two-dimensional icon, at the appropriate initial position. This validated the conversion and format utilities for the initialization files.

Next, the playback capability was created by using the data contained in the five converted post-processor files. Each file's information was loaded into separate arrays based on the clock time, in seconds, that the event occurred. Like the terrain features, a pop-

up menu is used to start the 'replay' action. When the playback option is activated, the program sets the elapsed time to zero. Then, as the elapsed time progresses, each of the array's clock times are checked against it. If an event matches the system clock, it is activated.

Minefields are drawn as rectangles, with the proper length, width, and orientation. Movement events are shown by simply moving the two-dimensional icon to its next position. Direct fire events are drawn with straight lines, with blue representing friendly fire and red representing enemy fire. Indirect fire events are depicted as circles. The edges of the circles are colored according to the side (red or blue), and the interior of the circle is painted black for high explosive or white for smoke. Smoke rounds remain on the map until their determined dissipation time. Destroyed vehicles are changed to a green color, and remain on the map at that position. An example of the playback program is shown in Figure 3.

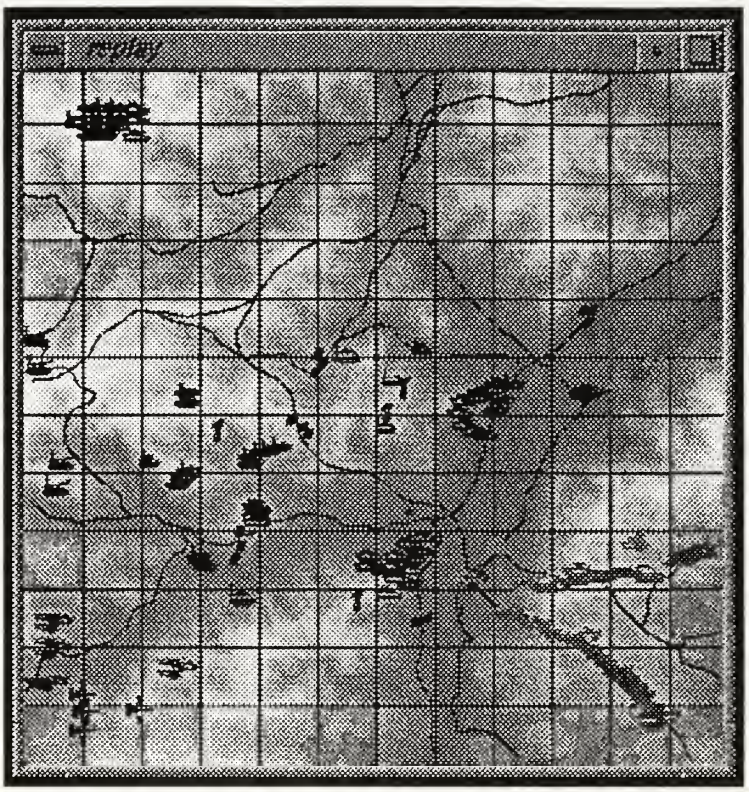

Figure 3: Fulda Gap with Roads and Scripted Battle.

The 'REPLAY' program proves that the conversion programs and display algorithms for the post-processor files are identical in every detail to the original scenario run on the actual JANUS(A) model. This confirmed, the next step was integrating JANUS(A) with the three-dimensional model NPSNET, as discussed in the next chapter.

IV. NPSNET: INTEGRATION

This chapter provides an overview of NPSNET and the methods used to integrate NPSNET with the JANUS(A) combat model. Topics discussed are the functions, programs and utilities required to create three-dimensional terrain skin for the JANUS(A) model within the NPSNET system. The procedures necessary to generate the triangular-mesh and the polygonized three-dimensional terrain, together with the methods used to create cities, trees and tree canopies are discussed. Also an algorithm to insure that trees and buildings are not drawn on rivers and roads is presented.

A. BACKGROUND

The Naval Postgraduate School's Computer Science Department created a system to render three-dimensional, real time simulations of military vehicles and terrain databases known as NPS Networked Vehicle Simulator (NPSNET). Specific attributes of the system include creating a three-dimensional, real time, virtual world at a cost of well under $100,000 [ZYDA 92]. Furthermore, it possesses the ability for quick adaptation with applications involving other modeling systems. In particular this chapter refers to the ongoing development of applications and interfaces for NPSNET with the JANUS(A) combat modeling system.

B. HARDWARE

NPSNET currently runs on a Silicon Graphics Incorporated (SGI), IRIS/4D family of computer systems utilizing the IRIX (UNIX based) operating system [SGI 91]. The entry level hardware for this model is the SGI Indigo-Elan with 16MB of memory and 100MB of available disk space. The cost for the minimal system is approximately $27,000.

C. DESCRIPTION

NPSNET utilizes two techniques to render terrain, using either triangular-mesh or discrete polygons as basic building blocks. The triangular-mesh terrain provides an efficient and simple terrain skin for the virtual world, but currently does not allow roads

and rivers to be drawn without significant tearing. In order to realistically provide the ability to render roads and rivers, NPSNET instead utilizes the discrete polygon (*polygonized*) terrain model [PRAT 92]. All objects (such as trees, vehicles and buildings) are rendered using a system known as NPSOFF [ZYDA 91a]. These stationary objects and polygonized terrain data are stored in files indexed by coordinates which match their corresponding terrain node (Terrain nodes are discussed in Section IV.G.). Moving objects are stored in an array assigned to each terrain node. Their positions are updated each time the graphics loop is executed [MACK 91].

NPSNET currently provides three modes of combat modeling. First, the model can be played in a networked, multi-user mode. In this mode, one or more players can chose and maneuver any vehicle in the three-dimensional virtual world. They can decide to fight with or against one another over the network. Additionally, a player can choose to fight a set of semi-automated forces. Players can view the world from the perspective of the vehicle commander or from an observer controller vehicle. The second modeling mode NPSNET uses is a script. With this method, the user can prepare scripted scenarios of combat operations and view the results in three-dimensions. The last mode includes receiving inputs over an Ethernet network from other three-dimensional combat models such as SIMNET. In this mode, NPSNET can fully interact with the other combat model.

D. NPSNET VEHICLE MOVEMENT AND SIMULATION

1. Vehicle Models

NPSNET provides an assortment of U.S. and foreign weapons system models. Currently, vehicle models range from U.S. M1A1 tanks and A-10 jet aircraft to Soviet made T72 tanks and BMPs. All of these vehicles are formatted and rendered using the NPSOFF system [ZYDA 91a].

2. Vehicle Data Structure

All information concerning the disposition of vehicles is stored in a data structure named *vehpostype*. This structure contains the information listed in Figure 4.

```
int   vehnum;              /*The vehicle identification number in NPSNET */
int   vehtype;             /*Type of vehicle model, an index into vehtypearray */
int   control;             /* Who is controlling it(i.e. script, network, etc.*/
int   undriven_control;    /* Who controls it when it's not the driven vehicle? */
int   gunfire;             /* Did it shoot in the last frame */
int   alive;               /* Vehicle is alive or dead */
float pos[3];              /* Vehicle position in space (i.e. 3D grid coordinates)*/
float direction,           /* X-Z plane orientation of veh in radians */
      viewdirection,       /* X-Z plane orientation of the drivers view in radians */
      elev,                /* Vehicles meters above ground (aircraft and observer) */
      gunelev,             /* Quadrant elevation of tube in degrees*/
      speed,               /* Meters per second */
      roll,                /* Degrees*/
      pitch;               /* Degrees*/
int jan_type;              /* Specifies janus symbol in 2d map */
int   behavior;            /* What type of behavior it is exhibiting*/
int   round;               /* Number rounds of ammunition vehicle has left*/
int   deadframes;          /* Number of frames has it been dead*/
long deadtime;             /* Time vehicle has been dead*/
int   coll_interval;       /* Minimum safe passing distance*/
float eye[3],              /* Location of the viewers eye in world coordinates.*/
      lookatpt[3],         /* Location of the point the viewer is looking at.*/
      lookfrompt[3];       /* Where viewer is looking when not in vehicle*/
float gas;                 /* Fuel remaining in vehicle*/
```

Figure 4: Vehicle Information Data Structure

Every vehicle present in a simulation is described with one of these *vehpostype* data structures, which is stored in an array named *veharray*[MAXNUMVEHICLES]. MAXNUMVEHICLES is a variable indicating the maximum number of vehicles that the simulation will allow -- currently 500.

The following discussion refers to the variables listed in Figure 4. The index for a particular vehicle in *veharray* is identical to *vehnum* for the vehicle. At initialization, each vehicle that is input into the simulation is assigned a sequential *vehnum* beginning with the number one. The observer controller vehicle has the number zero assigned as its *vehnum*. It is important to note that NPSNET uses a one dimensional array to store vehicle information. This requires only a single index to identify a unique vehicle in a simulation. JANUS(A), on the other hand, stores all of its vehicle information in a two-dimensional array, thus requiring two indices to identify a vehicle. A cross indexing routine, to allow the proper identification of a vehicle between the two systems, is discussed in Chapter V.

Vehtype is a variable that indicates which of the NPSOFF vehicle models is rendered for the particular vehicle. The *control* variable indicates whether or not the operator of the simulation is currently driving, thus controlling, the vehicle. If a vehicle is not being controlled by the operator, it is assigned the variable, *undrivencontrol.* The modes offered for an undriven vehicle include receiving commands from a script, receiving commands from the network, receiving commands through the semi-automated force system, or simply continuing at the same speed and direction that it currently has until the operator of the simulation changes it.

The variables that are of particular importance to the current disposition of a vehicle in the three-dimensional world are *pos*[3], *viewdirection, direction, elev* and *speed.* The variable, *pos*[3], is an array containing the grid coordinates for a vehicle. *Viewdirection* stores the angle of the turret (or angle of operators view) based on a coordinate system local to the vehicle itself. (Note: The X axis is considered the base axis from which all measurements begin and the positive direction of measure is clockwise.) The variable, *direction,* is the angle of the base of a vehicle with respect to the NPSNET terrain coordinate system. The *elev* variable indicates the number of meters a vehicle is above the terrain. Only aircraft and the observer controller vehicle can have an *elev* value greater than zero. The speed variable indicates the current speed of a vehicle, and is measured in meters per second in the NPSNET simulation.

The other variables that are listed in the *vehpostype* are important to the functioning of the NPSNET simulation but are not relevant to the translation of JANUS(A) data. Therefore, they will not be discussed.

3. Vehicle Activity in NPSNET

Every time the graphics loop is executed, each element in the *veharray* is updated. Based on the current speed and direction recorded for a vehicle, the new position is calculated and stored in the *pos*[3] variable. The vehicle is then rendered at this new location, with the orientation of the vehicle and turret prescribed by the *direction* and

viewdirection variables. These orientations can be changed by inputs from the user at the keyboard or with the space ball [MACK 91].

Other activities such as gunfire, or the destruction of a vehicle, are stored in the veharray for a vehicle. When the graphics loop updates the vehicle, it will render the appropriate response as indicated by the variables in the array. Activities of a very short duration, such as gunfire, are reset to *no gunfire* after the vehicle array is updated.

E. GENERATION OF NPSNET ELEVATION FILES

1. JANUS(A): Coordinate System.

NPSNET uses the JANUS(A) terrain database to construct the three-dimensional terrain. The JANUS(A) terrain database is merely a collection of grid coordinates with an elevation assigned to each coordinate. In order to store these elevations in a sequential file format, JANUS(A) divided the terrain into a grid. The spacing between the horizontal and vertical lines is known as the resolution value and is labeled "GAP" throughout the code. Each intersection is labeled by the cartesian coordinates of the horizontal and vertical line that make up the intersection. The *elevation*, *tree or city present flag*, and its respective *height factor* are recorded for each of these intersections. These grid intersections will be referred to as "checkpoints" in the remainder of this document. JANUS(A) references the cartesian coordinates for each checkpoint using its UTM coordinate as indicated in Figure 5. These UTM coordinates are translated to a local coordinate system where the lower-left hand corner of the map has the coordinates (0, 0) using Equation 4.1 and Equation 4.2

$$X_{\text{JANUS\_local}_i} = (X_{\text{UTM}_{\text{lower-left}}} - X_{\text{JANUS}_i}) \quad \text{(Eq 4.1)}$$

$$Y_{\text{JANUS\_local}_j} = Y_{\text{UTM}_{\text{lower-left}}} - Y_{\text{JANUS}_j} \quad \text{(Eq 4.2)}$$

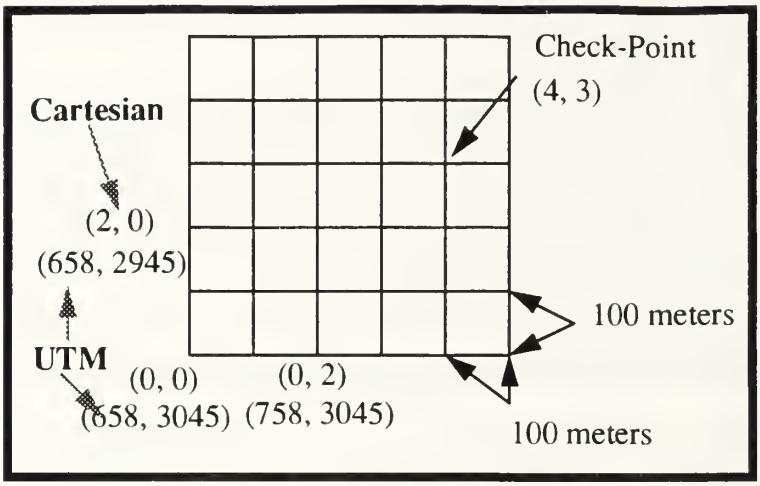

Figure 5: Description of JANUS Map Resolution

2. NPSNET: Coordinate System

Unlike JANUS(A), NPSNET utilizes a local coordinate system where the upper-left hand corner of the map is labeled (0, 0). Therefore, to allow NPSNET the ability to accurately reference the JANUS terrain database, each set of JANUS(A) coordinates are translated to reflect the NPSNET system. Since the ***x*** coordinate in the NPSNET coordinate system is the same as the x coordinate for the JANUS(A) *local* coordinate system, no further translation of the ***x*** coordinate is required. But because the ***y*** coordinates for the two systems are not the same each local JANUS(A) ***y*** coordinate is translated to match the NPSNET coordinate system. This translation is accomplished by subtracting each local JANUS(A) ***y*** coordinate from the total height of the map as written in Equation 4.3. (The *z* coordinate in three-dimensions is equivalent to the two-dimensional ***y*** coordinate. The remainder of the document will use the *z* coordinate when referencing the two-dimensional ***y*** coordinate.)

$$Z_{NPSNET_j} = \text{Map Height} - Y_{\text{JANUS\_local}_j} \qquad \text{(Eq 4.3)}$$

As explained in Chapter III, these translated coordinates, along with the elevations, road present, river present, city or tree present, micro-terrain roughness, and trafficability corridors for each checkpoint are stored in the file "*###.ele*".

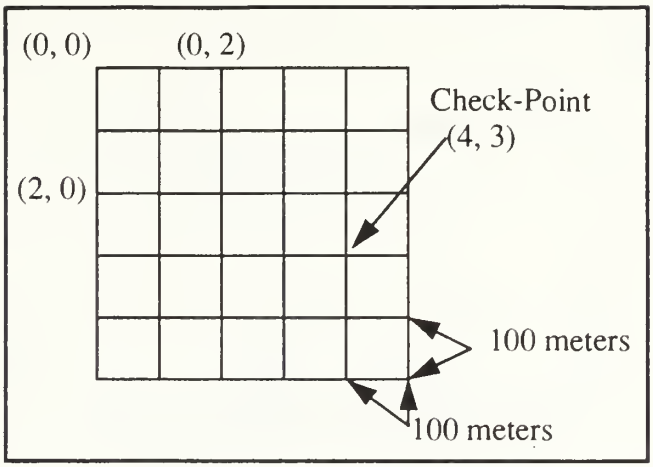

Figure 6: NPSNET: Coordinate System

F. GENERATING MESH TERRAIN

1. Creating the Binary Elevation file.

The binary elevation file is key to the creation of the three-dimensional mesh terrain, and the determination of the locations of all objects within the three-dimensional world. The program "*genbinaryelev.c*" extracts the elevation data from the file, *###.ele*. Then these elevations are then rewritten in column-major order (using the NPSNET cartesian coordinates as indices) to the binary file, *elev.bin.dat*.

2. Creating Mesh Terrain

Mesh terrain is created using the built in "IRIS" function for drawing a triangular mesh, commonly referred to as "t-mesh". Each checkpoint is treated as one of the three points of a triangle. The t-mesh terrain skin is built by moving from the left to the right of the map, then from the bottom to the top of the map. Figure 7 depicts the path of points used in the t-mesh function.

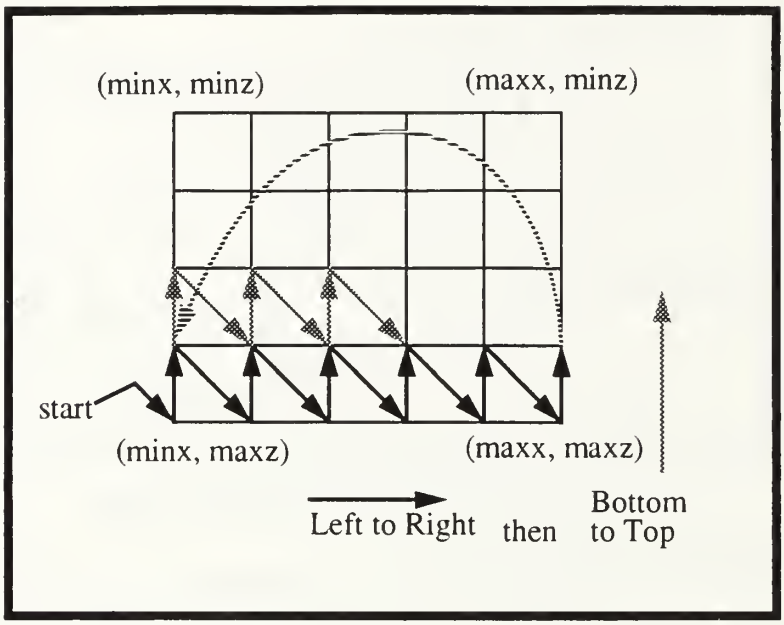

Figure 7: Tri-Mesh Terrain Traversal Pattern

When the right side of the map is reached, the mesh terrain algorithm moves the pointer back to the left edge and begins the next row. As stated earlier, the mesh terrain is more efficient and simpler to implement for the terrain model, but roads and rivers cannot be displayed on this terrain without significant tearing.

G. GENERATION OF POLYGONIZED TERRAIN SKIN

The best three-dimensional terrain image is rendered using a terrain skin comprised of discrete, triangular polygons. Because each polygon is a discrete element, each one can be manipulated -- or any object on it -- with great precision.

1. Structure of Polygonized Terrain

a. Description of Grid Node

A grid node is the basic building block for polygonized terrain (see Figure 8). Each node is made up of two triangles, an upper and a lower. (From the figure it appears that the positions of the two triangles are reversed, however, because measurements are

taken from the upper-left hand corner of the map, the *upper* triangle is the triangle farthest from the origin.)

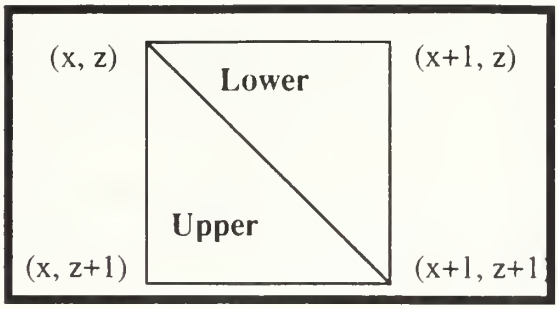

Figure 8: Basic Terrain Node

The numbering for each grid node uses the upper-left hand corner of the square as the local origin, with each corner of the node representing a checkpoint from the database. Each side of the *basic* grid node is equal to the resolution value (GAP).

b. Polygonized Terrain Resolution

The polygonized terrain is displayed with four degrees of resolution. The triangular polygons that make up the highest level of resolution have side lengths equal to the resolution value (GAP) of the database. The lower resolution polygons are larger (twice the size of the next higher resolution), thus, they are a less accurate depiction of the terrain. High resolution terrain is rendered as the set of polygons that are closest to the viewers look at point, while lower resolution polygons are rendered for terrain that is at increasingly greater distances from the viewers coordinates. Currently, high resolution polygons are used for terrain that is within a 1000 meter bounding box of the viewers x and z coordinates. Successively lower levels of resolution are used for polygons that are at distances greater than 2500, 4500 and 6000 meter, as measured from a bounding box that surrounds the viewer's position[MACK 91]. By using low resolution polygons for distant terrain, the number of polygons rendered is significantly reduced without sacrificing detail. This reduction increases the frame rate and efficiency of the machine.

c. *Description of a Terrain Quadtree*

Because each grid cell of the terrain is represented by four different resolution sets of polygons, a special data structure was implemented to store the data. This structure is called a quadtree -- a type of hierarchical data structure based on recursive subdivisions. The general idea behind quadtrees is that each level of the tree provides a more detailed description of an image or data. Nodes are either internal (with four children) or are terminal (with no children). They can be used to reduce storage space for data or to provide multiple resolution versions of data [MACK 91].

In order to get the transposed JANUS(A) database into the NPSNET quadtree format, the terrain is divided into blocks of cells. Each block is eight grid cells long and wide, and is marked as item "D" in Figure 9. This "D node" is the basic unit of storage for the terrain base, and is further subdivided into four nodes or children. These children are labeled as the set of "C" nodes. Each "C" node, too, is subdivided into four nodes labeled as the set "B". And finally each "B" node is divided into four "A" nodes. The length of the side of the block is eight times that of the GAP for the terrain. The corners of the upper and lower triangles are locally represented as (x, z), (x+8, z), (x, z+8) and (x+8, z+8). The resolution levels for the quadnodes are labeled as follows:

- HIGH 1 Side Length = GAP
- HIGH-MEDIUM 2 Side Length = GAP * 2
- LOW-MEDIUM 3 Side Length = GAP * 4
- LOW 4 Side Length = GAP * 8

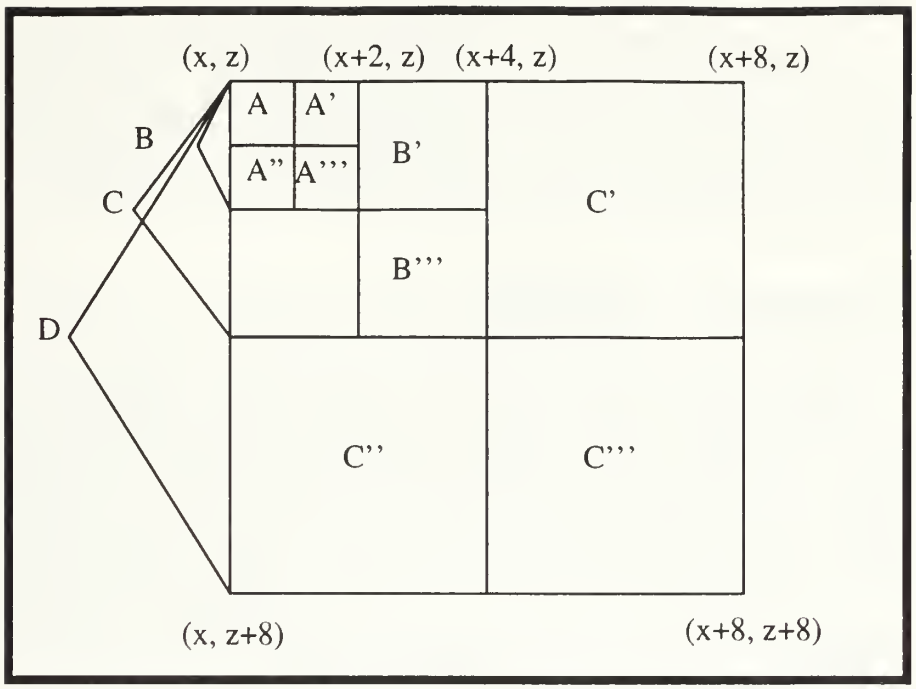

Figure 9: Quadtree: Terrain Illustration

These sets of nodes are stored in the quadtree structure, with each node or cell representing a node in the tree, and the four nodes that make up the subdivision of a larger node are children of the divided node (see Figure 10). The quadtree data structure produces a shallow tree that can quickly be searched to find the terrain cells required.

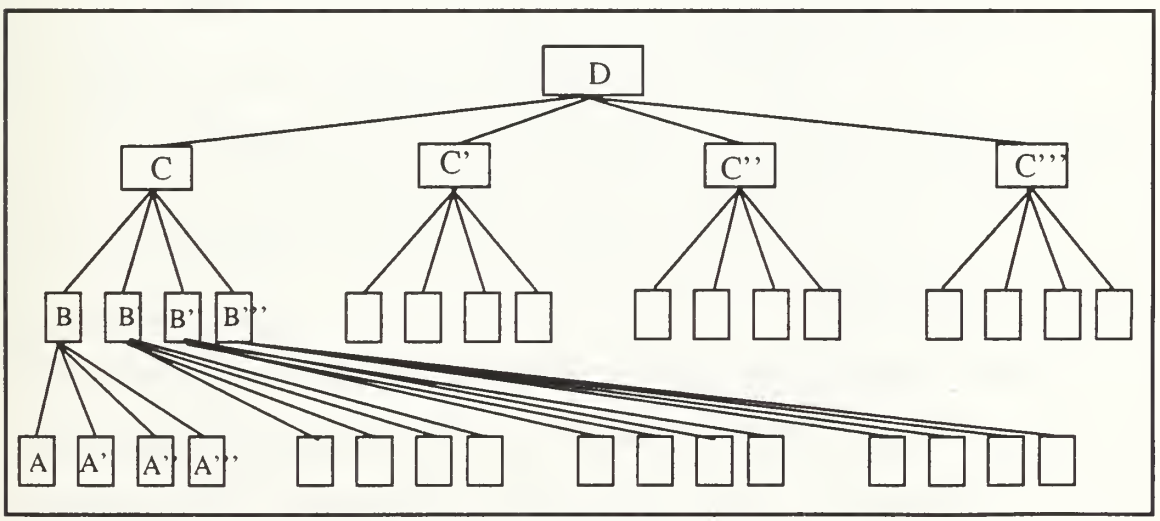

Figure 10: Quadtree: Data Structure Illustration

2. Generating Terrain Polygons

The program "*genblockcov.c*" divides the map into sets of quadtrees. In order to have a whole number of quadtrees in the database, the width of the entire map (XMAPSIZE) and the height of the map (ZMAPSIZE) must be a multiple of the product of the GAP value and the integer 8. (See Eq 4.3)

$$\text{Whole Number} = \frac{\text{MAPSIZE}}{(\text{GAP} \times 8)} \qquad \text{(Eq 4.4)}$$

If the map size, in either the *x* or *y* direction, does not provide a whole number, as indicated in Equation 4.4, then additional checkpoints need to be added to the database until Equation 4.4 is satisfied. The calculation to check for the appropriate number of checkpoints is conducted in the program "*readtrrn.c*". If more checkpoints are required, then the checkpoints are all given the elevation of the lower-left corner of the map. The following algorithm describes the additions of the checkpoints to the database.

```
/* Determine if extra checkpoints are needed to make */
/* the number of quadnodes a whole number. */
z_num_extra_nodes = (ZMAPSIZE / GAP) mod 8
x_num_extra_nodes = (XMAPSIZE / GAP) mod 8
if (x_num_extra_nodes != 0.0) then
   old_x_num_grids = x_num_of_grids
   /* Add the new x direction checkpoints */
   x_num_of_grids = x_num_of_grids +x_num_extra_nodes
if (z_num_extra_nodes != 0.0) then
   old_z_num_grids = z_num_of_grids
   /* Add the new z direction checkpoints */
   z_num_of_grids = z_num_of_grids +z_num_extra_nodes
/* Fill the elevation value of the new checkpoints */
for i = old_x_num_grids to x_num_of_grids loop
    for j = old_z_num_grids to z_num_of_grids loop
        grid(i, j).elevation = grid(0, 0).elevation
    end loop
end loop
```

Figure 11: Algorithm to Adjust Map Size According to Quadnode Requirements

Each set of data for a quadtree is stored in a unique file labeled *coverXXZZbin.dat*, where *XX and ZZ* reference the *x* and *z* indices for the quadtree. In order to optimize the search routines within a quadtree, each quadtree structure is represented as a heap in the code. A heap sort is used to find the correct resolution level within the file [MACK 91].

Each node of the quadtree is represented in the cover file as two sets of three points each, with each triple of points representing the upper and lower triangle of the node. With each point set, the quadtree indices, number of points in the set, the color for the triangle, an index indicating if the triangle is an upper or lower triangle, the normals for the triangle and the resolution level of the polygon are recorded. Figure 12 illustrates the storage pattern for a single node.

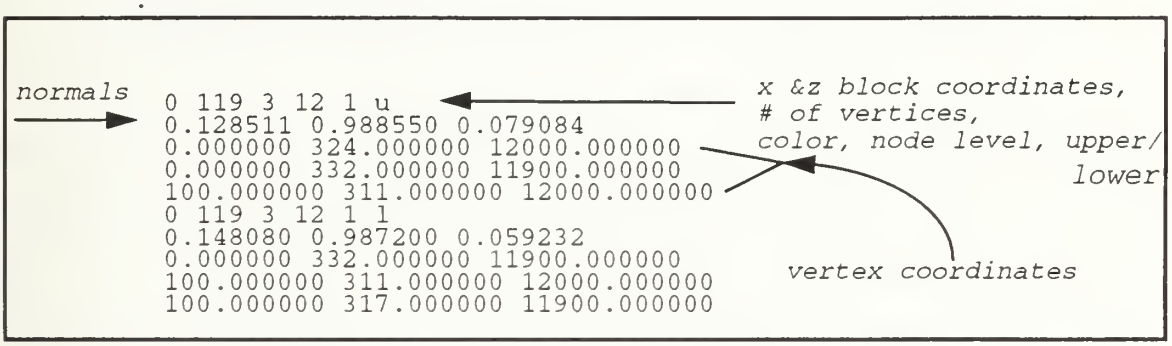

Figure 12: Quadnode File Representation

H. GENERATION OF NPSNET ROADS AND RIVERS

As stated in Chapter III, the program *readtrrn.c* creates the road and river data files labeled *road.dat* and *riv.dat.* These files list successive *x* and *z* pairs of grid coordinates for a road or river. A flag in the file indicates when a road or river ends and when the end of the entire file is reached. In order to create the road and river polygons, all road and river data must be grouped as pairs of successive points. Each pair of points is then converted into a multisided polygon with no more than six sides (six is the limit on the number of sides because that is the maximum number of sides created when a two triangles connect with a rectangle). The width of each polygon is standard -- currently 10 meters. The new road and

river polygons are subsequently written in the same format as each of the quadtree polygons, which is described in Figure 12. These road and river polygons are then sorted according to their coordinates, and placed in the quadtree file containing the corresponding coordinate. Polygon construction for roads and rivers is conducted using the program *makeroads.c*.

In the data set used in NPSNET, the polygons forming the road and river surfaces are coplanar with the polygons forming the underlying terrain surface. Therefore, the roads and rivers must be "decaled" onto the terrain. Decaling involves rendering the terrain twice, thus is more expensive to perform. For this reason, roads and rivers in NPSNET are only rendered on terrain at the highest resolution [MACK 91].

I. GENERATION OF NPSNET CITIES AND TREES

1. A Three-Dimensional Interpretation of Two-Dimensional Objects

Two-dimensional simulations traditionally render trees, cities, vehicles and other cultural objects as two-dimensional geometric shapes or icons (e.g. a square or circle). In the JANUS(A) model, trees and cities are depicted as colored squares. Trees and vegetation are colored varying shades of green, where the shade indicates the average height of the vegetation contained in the area of the square. Cities and urban areas are rendered in the same manner, using varying shades of yellow to depict height. The size of each square corresponds directly with the resolution level of the terrain database [JANU 86]. Developing a realistic three-dimensional representation from these two-dimensional squares proposed a problem.

As mentioned earlier, the NPSNET model currently uses NPSOFF models to render cultural objects such as houses and trees [ZYDA 91a]. Two problems arise when trying to use these models in conjunction with the JANUS(A) database. The first problem involves determining the appropriate number of single trees or buildings to render in any given grid cell. The second problem is the unacceptable degradation of the graphics frame

rate if the terrain is even moderately wooded. This is because the number of trees that need to be rendered significantly congests the graphics pipeline.

To solve the problems listed above, rendering of heavily wooded and large urban areas is performed with three dimensional canopies instead of individual trees or cities. The height of a canopy is taken from the URBAN and VEGETATION height factors extracted from the JANUS(A) terrain file. The length and width of each canopy is equal to the resolution value (GAP) of the database, therefore, each canopy covers exactly one high resolution grid cell. By rendering only one canopy per grid cell, the number of polygons used for the terrain model is greatly reduced, allowing an adequate frame rate in the model. To further increase the frame rate, the t-mesh function is used to draw the canopies.

The use of canopies solved the frame rate problem but did not provide a total solution to the problem of realism. If a grid cell contains a tree or city, together with a river or road, then the river or road would be covered by the canopy. The loss of the rivers and roads is not realistic and significantly degrades from the usefulness of the model. Therefore, a method was developed that stochastically rendered individual trees and buildings, using NPSOFF models, within a grid cell that contained a road or river plus a tree or city. However, this random placement of the objects causes another problem by allowing a tree or building to be placed in the center of a road or river -- again unrealistic. To eliminate this phenomena, another algorithm was developed that checks the placement of cultural objects against the location of rivers and roads. If they coincide, then the object is relocated. The generation of these cultural objects is discussed in the next sections.

2. Generation of City and Tree Canopies

In order to create the trees and cities, the information concerning the presence of trees or cities and the height of these cultural objects, that is stored in the file *###.ele,* is required. Its important to note that the tree and city data applies not only to the checkpoint of assignment, but also to the grid cell assigned to the checkpoint. Because each checkpoint makes up the corner of four different grid cells (except on the sides of the map), the user

must be careful to properly and consistently assign the correct cell to each checkpoint. If this assignment is inconsistent, then the user can easily lose track of which cell should have a tree or city drawn in it. The JANUS(A) convention of cell assignment is translated to match the NPSNET coordinate system and is used in the algorithms discussed next. This convention assigns each cell to the checkpoint in the northwest corner of the cell.

a. Description of a Canopy

Each canopy is composed of up to five t-mesh sides; a top, back, front, left, and right as depicted in Figure 12. Each of these sides is made up of two triangles. Two advantages are gained from this triangular design. First, it allows the use of the efficient SGI, t-mesh function. And second, the triangles that make up the top of the canopy correspond directly to the triangular polygons that make up the high resolution terrain polygon beneath the canopy. This correspondence allows the top canopy to fold in the center in the same fashion as the underlying grid cell providing a very smooth covering of the terrain.

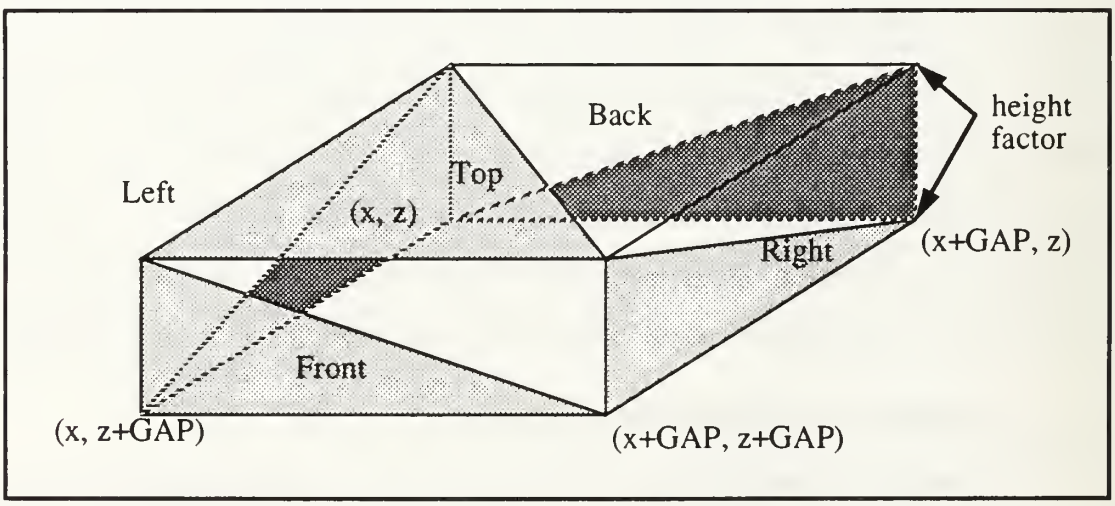

Figure 13: Drawing of a Tri-Mesh Canopy

b. Determining the Canopy Height

The height of each corner of the canopy is determined by the height factor assigned to the checkpoint that matches the corresponding corner of the canopy. This height factor for each corner of the canopy may be different, because the height factor for each corner of the cell is taken from the checkpoint directly beneath the canopy corner, not from the master checkpoint for the cell. If the checkpoint corresponding to a corner of the canopy does not contain a tree or city then the height factor is given the value of zero for that corner. The assignment of zero as a corner's height factor causes the canopy to slope into the ground.

c. Construction of Canopy Sides

Canopy sides are not always drawn. A side is drawn only if one of the following criteria are met:

- The cell, adjacent to the side in question, contains trees or cities and a road or river passes through the adjacent cell.
- The cell just north of the current grid cell does not contain trees or cities.
- The cell just west of the current grid cell does not contain trees or cities.

If one of the above conditions does not occur, then one of two other possible conditions exists. For these two conditions the side is *not* drawn. The first *non-draw* condition occurs when the cell adjacent to the side of the current grid cell, contains a tree or city *and* there is not a road or river in the cell. If this case occurs, the canopy top will smoothly connect with the canopy top of the adjacent cell. This decision to not render it reduces the number of objects in the graphics pipeline, and does not degrade the viewers image, because the side would not be visible anyway. The second non-draw condition occurs when there are no trees or cities in the cell adjacent to the southern or eastern side of the grid cell. As previously mentioned, this case causes the top of the canopy to slope to the ground.

In order to draw the sides or top of a canopy, the following pieces of data must be present:

- Elevation for each checkpoint.
- Flag indicating whether a tree or city is present in the cell.
- Point normals for the four checkpoints of the grid cell.
- Height factor for the tree or city.

The elevation, normals, and world coordinates for each checkpoint are stored in a two-dimensional array called *mapgrid*[i][j]. The indices, *i* and *j* are the NPSNET coordinates of the checkpoint. The city or tree present flag and the height factors for each checkpoint are stored in a separate, two-dimensional array called *treecityarray*[i][j], which is created by the program *maketrees.c*. A separate array is used to store the tree and city canopy data, allowing the NPSNET combat model to retain its flexibility for use with all prior and current modeling systems that use the NPSNET platform. The indices in both arrays are intentionally the same because they refer to the same grid intersection.

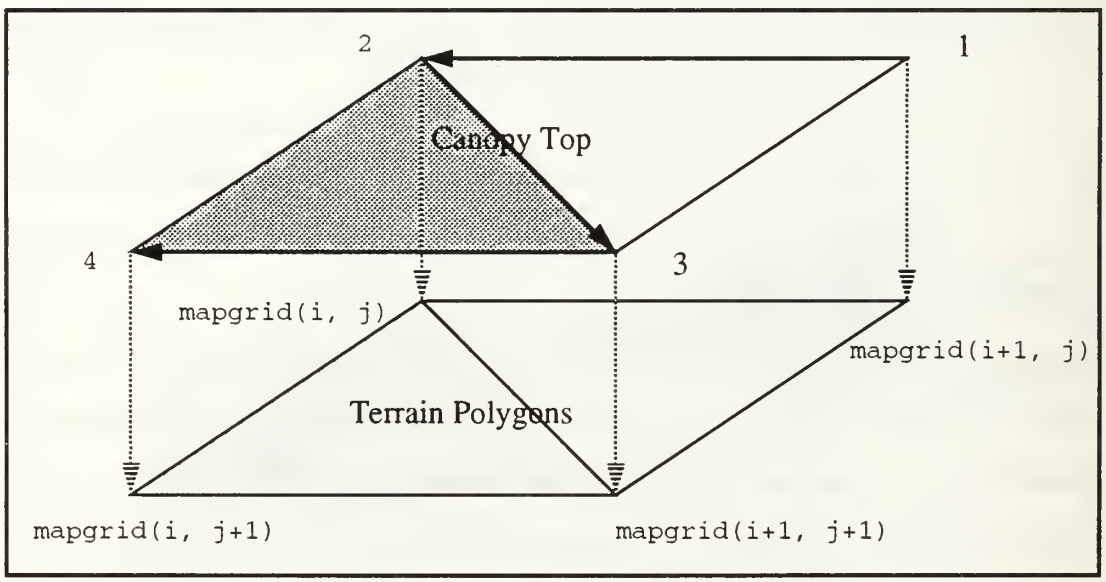

Figure 14: Order of Points for T-Mesh Canopy Top

The order of points for the t-mesh function is depicted in Figure 14. This ordering was chosen to ensure that the direction of the hypotenuse of the triangles in the canopy matched the direction of the hypotenuse for the polygons used in the underlying

terrain skin (i.e. the hypotenuse runs from the northwest corner to the southwest corner of the cell). The algorithm to draw the canopy top follows:

```
tree_data: is an array of float numbers that contain the x, y
and z coordinates of a point.
 /* point #1 */
 tree_data[0] = x;
 tree_data[1] = mapgrid[i][j+1].elev + tree_data[2] = z + GAP;

 treecityarray[i][j+1].height;
  n3f(mapgrid[i][j+1].normal);
  t2f(blocktreetexutre[0]);
  v3f(tree_data);
 /* point #2 */
 tree_data[0] = x;
 tree_data[1] = mapgrid[i][j].elev + treecityarray[i][j].height;
 tree_data[2] = z;
  n3f(mapgrid[i][j].normal);
  t2f(blocktreetexture[3]);
  v3f(tree_data);
 /* point #3 */
 tree_data[0] = x + GAP;
 tree_data[1] = mapgrid[i+1][j+1].elev +treecityarray[i+1][j+1].height;
 tree_data[2] = z + GAP;
 n3f(mapgrid[i + 1][j + 1].normal);
 t2f(blocktreetexture[1]);
 v3f(tree_data);
 /* point #4 */
 tree_data[0] = x + GAP;
 tree_data[1] = mapgrid[i+1][j].elev + treecityarray[i+1][j].height;
 tree_data[2] = z;
 n3f(mapgrid[i + 1][j].normal);
  t2f(blocktreetexture[2]);
  v3f(tree_data);
  endtmesh();
  if(flags.textureflag) texbind(TX_TEXTURE_0,0);
```

Figure 15: Algorithm to Draw the Canopy T-Mesh.

The functions `n3f, t2f` and `v3f`, listed in Figure 15, are SGI library functions that bind the normals, texture and grid coordinates to a particular point. The t-mesh operation uses these values to render the mesh. The sides of the canopies are drawn using an algorithm similar to the one written for the canopy top (see Figure 14). However, the checkpoints and heights vary for each of the sides. The ordering of the points for the side walls, used as input to the t-mesh function, must also cause the formation of two distinct triangles.

The normals for the walls are as follows:

- back wall normal (0, 0, -1)
- front wall normal (0, 0, 1)
- right wall normal (1, 0, 0)
- left wall normal (-1, 0, 0)

3. Placement of Individual Trees and Buildings

When a road or river is present in a grid cell containing a tree or city, individual trees or buildings are rendered in lieu of canopies. The general description of a tree and building is illustrated in the figure below.

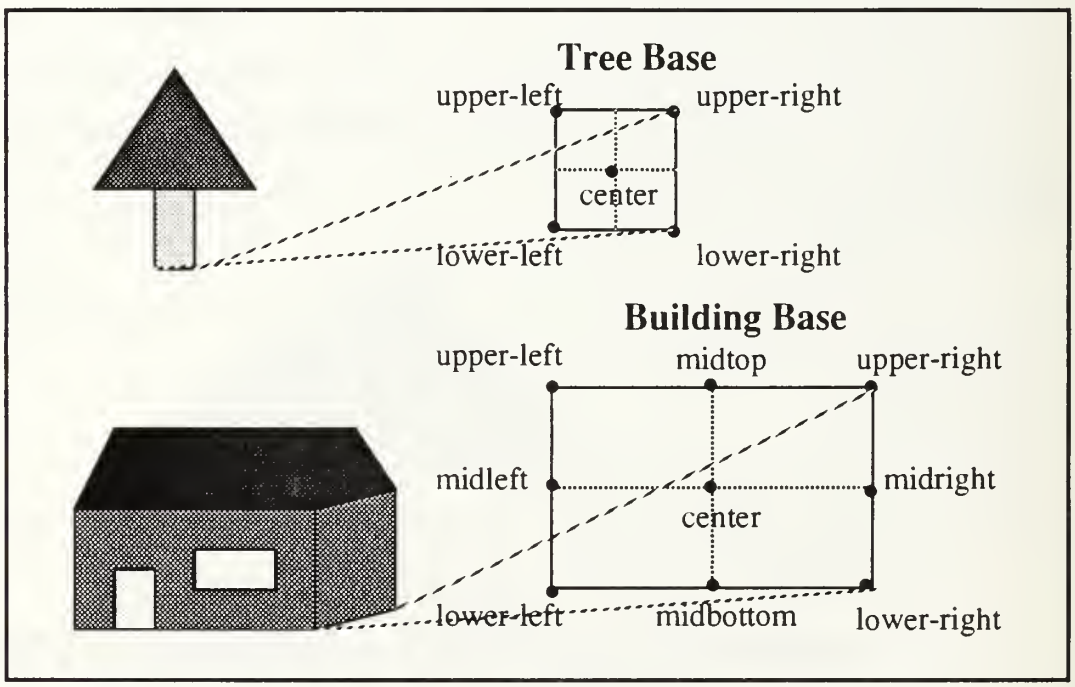

Figure 16: Cultural Object Description

The number of trees or buildings placed in a grid cell is based on the height factor assigned to the cell from the JANUS(A) data base. The exact location of each of these cultural objects within a grid cell is determined randomly. The following routine ensures that a tree or building is not placed on a road or river. It accomplishes this by ensuring that

each of the vertices of the base of the object, as depicted in Figure 16, do not fall inside a polygon that makes up a road or river.

a. Tree Placement

To determine whether or not a vertex falls inside a polygon, the following six-step Tree Placement Algorithm was developed:

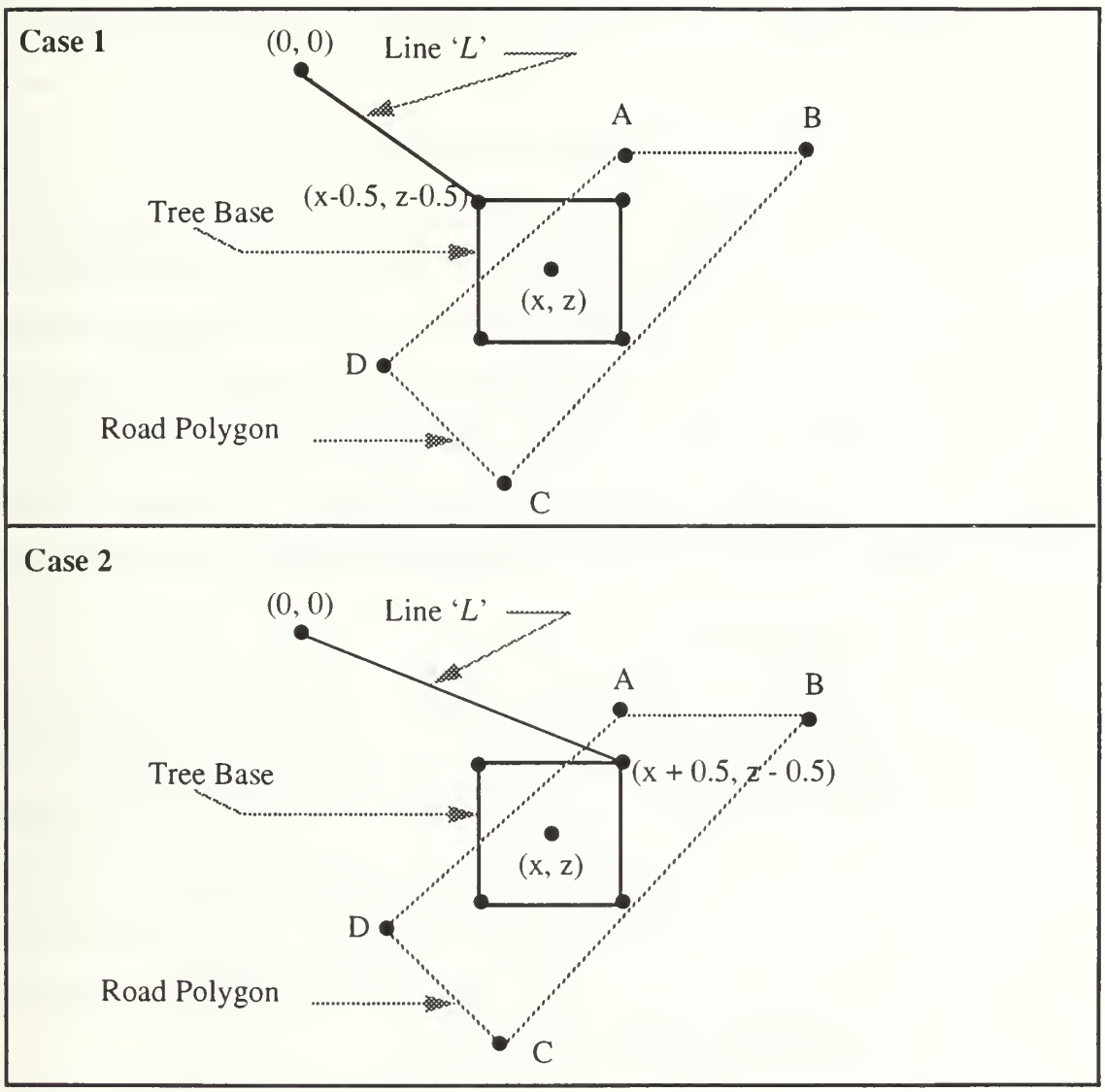

Figure 17: Polygon Intersection Examples

(1) Create a new point at the location (0, 0) on the map.
(2) Then calculate the equation of the line, 'L', that passes through the tree base vertex in question and the point (0, 0) using Equation 4.7.
(3) Determine the equations for the lines that make up each side of the road or river polygon (line Segments AB, BC, CD, DA as shown in Figure 18.).
(4) For each line segment of the road or river polygon, test if it intersects the line segment 'L'.
(5) Count the number of polygon line segments that intersect the line 'L'. If the count is even then the vertex is not located inside the road or river polygon. If the count is odd the vertex is located inside the road or river polygon.
(6) If the segment intersection count is odd, then reposition the tree, and test for an intersection again.

Figure 18: Tree Placement Algorithm

In case 1 of Figure 17, the line segment *L* does not intersect any of the line segments that make up the road or river polygon. Therefore the line segment intersection count is zero. Since 'zero' is an even number, vertex (x - 0.5, z - 0.5) does not lie within the road/river polygon. In case 2, line segment *L* intersects line segment *DA* and no others. Here, the line intersection count is one. Because the number 'one' is an odd number the vertex (x + 0.5, z - 0.5) does lie within the road or river polygon. Therefore this tree must be relocated and tested once again for possible intersection with a road or river polygon.

b. Building Placement.

The placement of the buildings uses the tree placement algorithm with the following modification. Because a building is wider than a road it is possible for all four corners of the building base to lie outside of a road or river polygon, while the middle portion of the building lies inside. Therefore, the tree placement algorithm is modified to check the midpoints of each line segment that make up the base of the building (see Figure 16). This guarantees that if a portion of a building is set on a road or river, then one of the vertices or midpoints will be located in a road or river.

c. *Equations for Determining Line-Polygon Intersections*

The tree placement algorithm is straightforward, but the equations that determine line segment intersections deserve further mention. First, the following equations are used to calculate the equations of the various line segments.

$$\text{Line Slope: } m = \frac{y_j - y_i}{x_j - x_i} \qquad \text{(Eq 4.5)}$$

$$\text{Y Intercept: } b = y_j - (m \times x_j) \qquad \text{(Eq 4.6)}$$

$$\text{Line Equation: } y = mx + b \qquad \text{(Eq 4.7)}$$

To determine whether the line segment *L* intersects any of the line segments of the road or river polygon, the following geometric principles are required. (The following discussion refers to Figure 19.) Chose the points P_1 P_2 and P_3 and the points P_1, P_2 and P_4, where P_1 and P_2 are the end points for line segment A, and P_3 and P_4 are the end points of line segment B. If the radial direction of the ordering of the points, as listed above is not the same in both patterns, then the line segments must intersect with each other. Only the intersection of the line segments would cause one of the end points from line segment B to fall in-between the endpoints of line segment A, thus causing the ordering of the two sets of points to change form clockwise to counterclockwise or vice versa.

The same is true about the following two orderings of the points P_3, P_4 and P_1 and the points P_3, P_4 and P_2. This leads to the following proposition:

> A necessary and sufficient condition for two closed line segments to cross each other is:

$$[\text{order}(P_1, P_2, P_3) \neq (\text{order}(P_1, P_2, P_4))] \wedge [\text{order}(P_3, P_4, P_1) \neq \text{order}(P_3, P_4, P_2)] \qquad \text{(Eq 4.8)}$$

[KANA 91]. As pointed out in the paper by Y. Kanayama, on page 8 of his article, the signs associated with the areas of the triangle formed by the triples of line segment end points, correspond directly with the ordering of the points. Therefore, Equation 4.10 is easily implemented using the areas calculated with Equation 4.9.

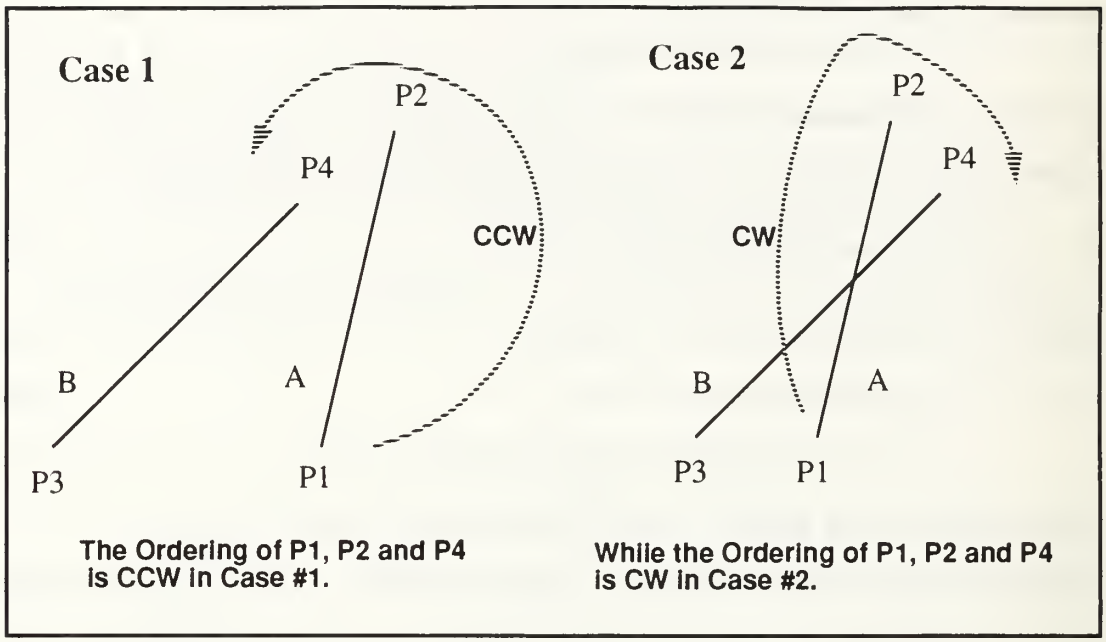

Figure 19: Illustration of Line Segment Intersections

The ***order*** function returns the sign (positive, negative, or zero) that is assigned to the area of the triangle formed by the triple of points that are input to the function. The number 'one' indicates a clockwise ordering, and the number 'negative one' indicates a counterclockwise ordering. The number 'zero' indicates that the points are coincident upon the same line segment.

$$Area\,(P_1, P_2, P_3) = (\frac{1}{2}) \times (((x_2 - x_1) \times (y_3 - y_1) - ((x_3 - x_1) \times (y_2 - y_1)))) \quad \text{(Eq 4.9)}$$

$$Order\,(P_1, P_2, P_3) = \begin{cases} (1) \text{ if sign } (Area\,(P_1, P_2, P_3)) > 0 \\ (0) \text{ if sign } (Area\,(P_1, P_2, P_3)) = 0 \\ (-1) \text{ if sign } (Area\,(P_1, P_2, P_3)) < 0 \end{cases} \quad \text{(Eq 4.10)}$$

V. NPSNET: GENERATION OF A THREE-DIMENSIONAL SCRIPT

This chapter outlines the procedures required to create a script for the three-dimensional simulation, NPSNET, from postprocessor files written by the two-dimensional, JANUS(A) simulation. The major hurdle to overcome for this conversion, is that a two-dimensional game does not require (and in this case, does not produce) all of the information needed to define what a vehicle is actually doing at *every* moment in the game.

A. THREE-DIMENSIONAL INFERENCE PROBLEM

In the traditional JANUS game, vehicles are represented on a two-dimensional screen using a simple icon. Depending on whether a vehicle is classified as "friend" or "foe", this icon is permanently faced to the left or right and no indication is given to its direction of travel, its orientation, or its gun tube/sight orientation [JANU 86]. As mentioned in Chapter III, JANUS(A) supplies the following five types of postprocessor files:

1) Movement (position of a vehicle).

2) Artillery (location of indirect fires).

3) Fires (which vehicle shot who).

4) Kills (who was killed by whom).

5) Detection (which vehicle detected another vehicle) [JANU 86].

In general these files list the x and y coordinates of a vehicle at the time an event occurs. Additionally, who it shot, the location of the target, the location of impact for each artillery shot, and the clock time for each of these events is included with these listings [JANU 86]. When the actual JANUS model is running, it is possible, upon request, to find out the speed of a vehicle and its line-of-sight fan. However, this information is <u>not</u> recorded in the postprocessor files.

In order to recreate a viable three-dimensional representation of the game, new information about what each vehicle was doing between recorded events must be inferred. The *C Language Integrated Production System* (CLIPS) expert system shell was chosen as

the inference engine to generate this new information [GIAR 89]. CLIPS was chosen because of the considerable amount of pattern matching that needs to take place.

B. APPROACH

The first step in the information inference process is to determine the data that NPSNET requires for rendering. One of the capabilities of NPSNET is to display and move vehicles according to a script. To read the script, NPSNET first compares the current game clock with the time assigned to the top element in the script file. If the game clock is greater than or equal to the event time, then the event is read and its instructions are implemented. To simplify the processing of a script, each line in the script file is considered an event. The information contained in each event line is as follows:

- Time
- Vehicle Number
- Vehicle Type
- X coordinate
- Z coordinate (referred to as the Y coordinate in a 2D world)
- Vehicle orientation (in positive radians)
- Weapon orientation (in positive radians)
- Speed (meters per second)
- Shot fired (1= Yes; 0 = No)
- Alive (1= alive; 0 = killed)
- Elevation (ground vehicle = 0; helicopter = 100 meters; plane = 200 meters)

After NPSNET reads an event, it assigns the event speed and direction to the three-dimensional vehicle model. If the vehicle shoots or dies, a flash or burning hulk is rendered respectively.

C. CREATING THE NPSNET SCRIPT FILE

The key to inferring new information from the post processor files is to compare the current event with the chronologically subsequent event for the same vehicle. From these comparisons, many useful pieces of information, such as the speed and direction of

movement can be inferred. Currently, events in the JANUS postprocessor files are listed chronologically, with a postprocessor file appearing as shown in the following table [JANU 86]:

TABLE 4: MOVE POST-PROCESSOR FILE EXAMPLE

TIME	Vehicle Side	Vehicle Number	X Coord	Y Coord
0.10	2	5	345.8	5678.9
0.12	1	12	578.5	5478.2
0.20	1	8	566.1	5410.0
1.28	2	20	367.2	5554.9
1.30	1	12	577.1	5465.7

In order to compare two events for a single vehicle (e.g. Vehicle 12, Side 1 from Table 4), there needs to be an easy way to locate the next event for a particular vehicle. Searching each of the five postprocessor files for chronologically subsequent events is a memory intensive process that would quickly become time consuming and thus not useful. Therefore, an event file is created for each vehicle. Next, each of the JANUS postprocessor files is read and the events pertaining to each individual vehicle is stored in its own file. Furthermore, a letter is appended to the head of each event to identify its event type (fire, move, etc.). A typical vehicle event file appears below:

TABLE 5: EVENT FILE FOR VEHICLE 12, SIDE 1

Letter	Time	X Coord	Y coord	Target X coord	Target Y coord
M	0.12	578.5	5478.2		
M	1.30	577.1	5465.7		
F	2.88	570.5	5461.2	458.9	5889.7
K	3.5	570.3	5459.9		

M= *move*; F = *fire*; K = *kill*; A = *artillery*; D = *detection*

With the events for each vehicle stored in a single file, it is now easy to read an entire vehicle event file into memory, compare two chronological events for a vehicle, and infer some useful information about the vehicle's activities between listed events.

D. INFERRED INFORMATION REQUIREMENTS

In general, there are three activities that a vehicle could possibly perform between listed events. It can be *halted*, *moving*, or *killed*. In order to present a realistic three-dimensional representation of these three activities, the following pieces of information are inferred:

- Orientation of the vehicle
- Orientation of the weapon (turret)
- Direction of movement (if moving)
- Speed

The *basic* speed and orientation of the vehicle and weapon are obtained by performing the simple calculations described below. The general mission of the vehicle is determined in order to gain a more refined estimate of its weapon's orientation and speed. These calculations are implemented using CLIPS Object-Oriented Language, (COOL) [GIAR 91]. COOL is used because the encapsulation of the information pertaining to one object -- and there are a lot of information slots per object -- makes it easy to manipulate the object as a whole.

E. BASIC CALCULATIONS

These simple calculations are used to determine basic speeds and directions.

Speed: The classic equation:

$$velocity = (\Delta d) / (\Delta t) \qquad \text{(Eq 4.11)}$$

The change in distance is calculated using the equation for the length of a line when two points are known:

$$\Delta d = \sqrt{(x_{next} - x_{current}) + (y_{next} - y_{current})} \qquad \text{(Eq 4.12)}$$

The change in time is calculated:

$$\Delta t = t_{next} - t_{current} \qquad \text{(Eq 4.13)}$$

Orientation (θ):

To determine the angle, θ, take the arc tangent of the quotient of the difference between the y coordinates and the x coordinates.

$$\text{angle:} \quad \theta = \arctan \frac{(y_{next} - y_{current})}{(x_{next} - x_{curent})} \qquad \text{(Eq 4.14)}$$

Once θ is determined, then the quadrant (using the Cartesian Coordinate System) is found by analyzing whether the difference in the x and z coordinates are positive or negative.

Then these values are used to access the following table, which gives the quadrant factor. This factor is added to θ to produce the actual orientation of the vehicle (All angles are measured using the x axis as the base or zero axis, with clockwise as positive rotation. All calculations are in radians.).

TABLE 6: ANGLE OF ORIENTATIONS QUADRANT FACTORS

sign x coord	sign z coord	quadrant factor
+	+	0.0
+	-	2π
-	-	π
-	+	π

F. THE HEURISTIC MODEL

There are two types of missions that a vehicle may be assigned. They are the classic defensive or offensive missions. Determining which mission a vehicle is assigned assists in the inference of information between events. The different heuristics are explained next.

1. Defensive Heuristic

There are several characteristics that suggest that a vehicle is in a defensive posture. The primary or tell-tale defensive characteristic is when a vehicle is travelling at a speed of less than 3 k.p.h. This speed tends to suggest that the vehicle is either on an extremely slow and deliberate offensive mission or it is really sitting still (in a defensive position) for most of the time interval between the two listed events. If it moves to its next position at a more deliberate speed of about 13 -15 k.p.h., it is assumed that it is moving to an alternate defensive position. If this extremely slow speed is encountered, there are two other pieces of evidence that, if present, tend to confirm the defensive hypothesis. One of these additional pieces of evidence is whether the vehicle just completed firing a shot or will fire a shot in the future. This tends to be true because a vehicle shooting on the move would normally be travelling at a speed much in excess of 3 k.p.h. if it wants to survive. The second piece of evidence assumes that if a vehicle fires in the future, and if the distance to the next firing position is less than 100 meters, it is probably moving between fighting positions and hence is in a defensive posture. If all of the above conditions are met then the program asserts that the vehicle is in the defense, and as a consequence halts the vehicle and orients it towards the enemy (towards the next target location). Then, the vehicle moves to its next event location at a set speed of 13 k.p.h., ensuring that it arrives at the next event location on time. There are certainly other heuristics that can suggest if a vehicle is in the defense or not; they currently are not implemented.

A second tell-tale heuristic is whether a vehicle does not move for over five minutes. Not moving for over five minutes suggests that the vehicle is in a defensive

posture -- viewing the engagement area. Then at a certain time it would move quickly to its next defensive position.

2. Offensive Heuristic

Its assumed that if a vehicle's events do not confirm the defensive heuristic then the vehicle is considered to be on an offensive mission. This is a simple but rather accurate hypothesis but does not exhaust the need for other inferred knowledge.

3. Speed Smoothing Heuristic

An anomaly that becomes very apparent in a three-dimensional world, that is not readily observed in the two-dimensional JANUS(A) simulation is, at times, a vehicle travels at a speed in excess of 45 m.p.h. Unless the vehicle is an aircraft, this is very unrealistic, and probably due to minor modifications in input by the user at the time of the original JANUS(A) run. This anomaly often occurs between the initial position for a vehicle and the first event listed in the postprocessor files.

If these two conditions occur, then at start time, the vehicle is rendered at the position specified by the first movement rather than the location specified by the initial conditions. It is also given a speed of zero. If this *excessive speed anomaly* does not occur between the initial position and the first event, then the speed is smoothed.

The smoothing algorithm slows the speed of the vehicle to 25 k.p.h. This decrease in speed has a natural side effect. It will cause the vehicle to not be at the correct location when its next event is scheduled to occur (call this upcoming event, *A)*. To compensate for this problem a new MOVE event is created (event *B*) which will keep the vehicle moving at a speed of 25 k.p.h. to the location that was specified in event *A*. In order to get the vehicle back in synchronization with the remainder of the event list, the speed associated with moving from event *A* to its subsequent event is increased to allow the vehicle to arrive at the follow-on event's location on time.

4. Vehicle Orientation Heuristic

As a rule, the orientation of the vehicle, if moving, is always in the direction of travel. If the vehicle is halted, the orientation depends on the following pieces of evidence:

(1) If the vehicle is firing during the current event then orient towards the target.
(2) If the vehicle is not firing but will fire in the future, orient the vehicle towards the next target location.
(3) If the vehicle is not firing and will not fire in the future, then use the vehicle orientation from the last event as the current orientation.
(4) If the vehicle is placed on the battlefield but does not have any events associated with it, then orient the vehicle in its initial view direction.

5. Weapon Orientation Heuristic

The heuristics for weapon orientation are similar to those used for the vehicle orientation with a few differences.

(1) If the vehicle is firing during the current event then orient the weapon towards the target.
(2) If the vehicle is not firing but will fire in the future, orient the weapon towards the next target location.
(3) If the vehicle is not firing, will not fire in the future, but did fire in the past then use the weapon orientation from the last firing event as the new weapon orientation.
(4) If the vehicle never fires then orient the weapon in the same direction as the vehicle.

G. CALCULATION OF THE HEURISTICS IN CLIPS

Using the CLIPS Object Oriented Language (COOL), each of the events listed in a vehicle event file is read into one of the following objects: *MOVE, FIRE, KILL, ARTY*. It should be clear as to which type of event is stored in which object [GIAR 91]. After the entire file is read into memory, then each event is compared against the next event or an upcoming firing event. These comparisons generate facts that are stored in the form of (condition, <fact>). Then, based on the facts generated, CLIPS' inference engine will choose the appropriate rule to execute. These *rules* call *methods* and *functions* which

generate the data required for NPSNET [CLIP 91]. Table 7 lists the combinations of facts that CLIPS uses in determining the correct rule to use. Once a rule is chosen and the NPSNET event data is generated, this data is written to a file specified for the current vehicle.

Once the event file for all of the vehicles is evaluated, each of the files containing the heuristic data is combined into a single event list file and labeled *janusscript.dat*. This single file is then sorted chronologically. After sorting it is ready to be used by NPSNET as a script file.

TABLE 7: VEHICLE STATE CONDITIONS

Primary activity	Prior - shot	Post -shot	Shot now	speed > 45 kph	3kph < speed < 45	speed < 3kph	speed = 0.0	kill	Dist to fire <.1km	Dist to fire >.1 km	Time to next event > 5 min
Halt	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	DC
Halt	no	no	no	no	no	no	yes	no	DC	DC	DC
Halt	yes	no	no	no	no	no	yes	no	DC	DC	DC
Halt	DC	yes	no	no	no	no	yes	no	DC	DC	DC
Halt	DC	DC	yes	no	no	no	yes	no	DC	DC	DC
Killed	DC	DC	DC	DC	DC	DC	DC	yes	DC	DC	DC
Move	no	no	no	yes	no	no	no	no	no	no	DC
Move	DC	yes	no	yes	no	no	no	no	DC	DC	DC
Move	DC	DC	yes	yes	no	no	no	no	DC	DC	DC
Move	no	no	no	no	yes	no	no	no	DC	DC	DC
Move	yes	no	no	no	yes	no	no	no	DC	DC	DC
Move	DC	yes	no	no	yes	no	no	no	DC	DC	DC
Move	DC	DC	yes	no	yes	no	no	no	no	no	DC
Move	no	no	no	no	no	yes	no	no	no	no	DC
Move	yes	no	no	no	no	yes	no	no	no	no	DC
Move	DC	yes	no	no	no	yes	no	no	no	yes	DC
Move	DC	yes	no	no	no	yes	no	no	yes	no	yes
Move	DC	yes	yes	no	no	yes	no	no	no	yes	DC

TABLE 7: VEHICLE STATE CONDITIONS

Primary activity	Prior - shot	Post -shot	Shot now	speed > 45 kph	3kph < speed < 45	speed < 3kph	speed = 0.0	kill	Dist to fire <.1km	Dist to fire >.1 km	Time to next event > 5 min
Move	DC	yes	yes	no	no	yes	no	no	yes	no	yes
Move	DC	no	yes	no	no	yes	no	no	no	no	yes

Note: Not every permutation of conditions was considered meaningful. The code DC (don't care) is used to identify conditions that are not meaningful.

VI. NETWORKING AND REAL TIME SIMULATION

A. JANUS(A) FOR THE UNIX OPERATING SYSTEM

1. Background

The U.S. Army realized the advantages of running the JANUS(A) combat model on the popular and compact UNIX based workstations instead of on the cumbersome VAX/ Tektronix systems currently in use. In August of 1991, TRAC Monterey contracted with the Rand Corporation to develop a version of JANUS(A) that would operate on a Sun/ UNIX workstation. Jim Guyton, of Rand, delivered a working prototype of JANUS(A) for UNIX in April 1992 [GUYT 92].

The goal, and most exciting part of this research, was the question of developing a real-time, network link between the traditional JANUS(A) model and NPSNET. In the past, this was not feasible because of the drastically different protocols and platforms. With the introduction of the new version of the JANUS(A) combat model, it finally became possible to construct this real-time network.

2. Contrasts Between the UNIX and VAX Versions of JANUS(A)

This new UNIX-based version of JANUS(A), which is called JANUS(X), does not change the "inner-workings" of the original combat model. The model itself is identical in both versions, in fact, the same FORTRAN code is used in both models. This reuse of the VAX-FORTRAN code is made possible by the Sun 4-FORTRAN compiler, which is capable of translating VAX data types and system calls into their UNIX equivalents [SUN 91].

The only viable difference between the two versions of JANUS is that the UNIX version bypasses all FORTRAN calls to the Tektronix screens, and replaces then with the 'C' programming language calls to the X-Windows environment [GUYT 92]. Also, much like the converted files mentioned in Chapter II, the input data, such as the force and system

files, are translated into a file format that is compatible with the UNIX operating system [GUYT 92].

3. JANUS(X) Networking Capabilities

As mentioned above, the VAX-version of JANUS(A) is hindered by its antiquated hardware and non-networking capabilities. This version can only be displayed on Tektronix monitors, with the Blue force on one screen, and the Red force on another as shown in Figure 20.

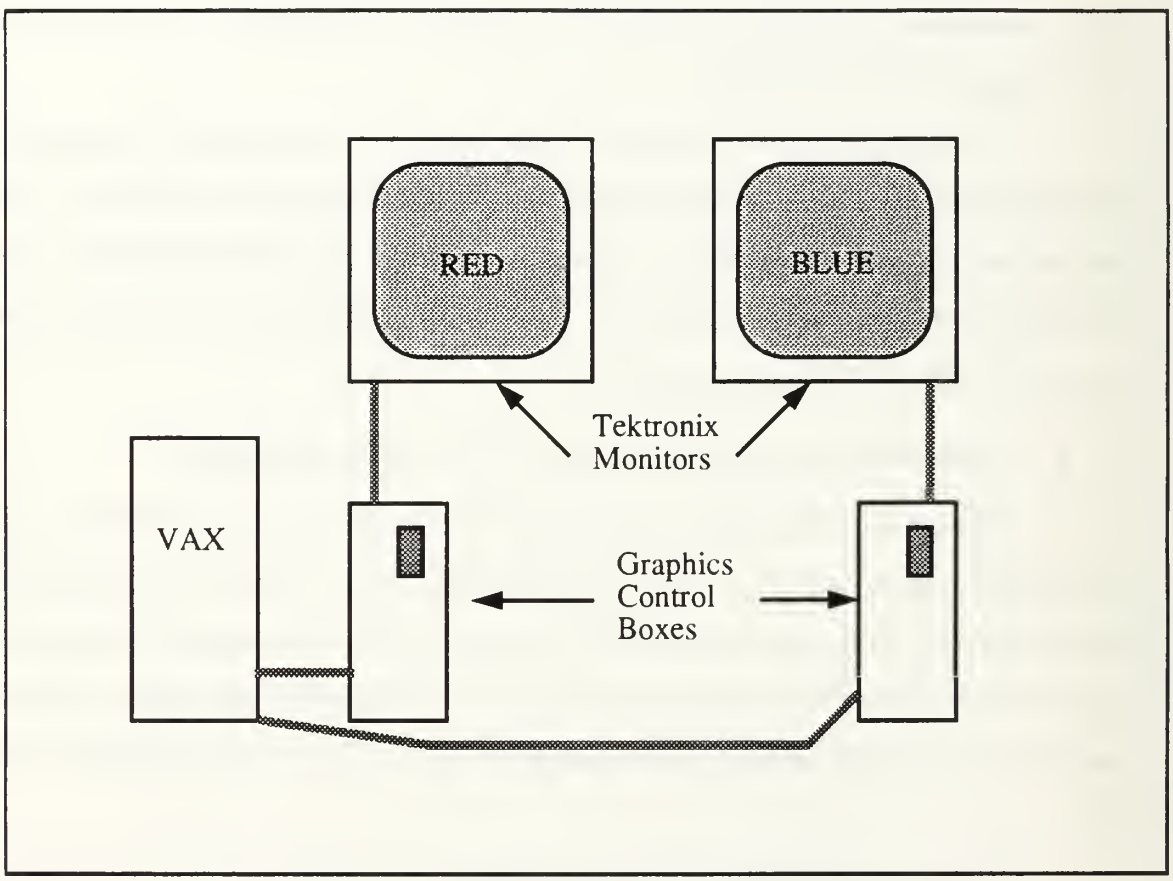

Figure 20: The Traditional JANUS(A) Hardware Setup.

JANUS(X), on the other hand, has the added flexibility of being run on a Sun Workstation with the display piped to the same monitor, or any other workstation and monitor with X-Windows capability as shown in Figure 21.

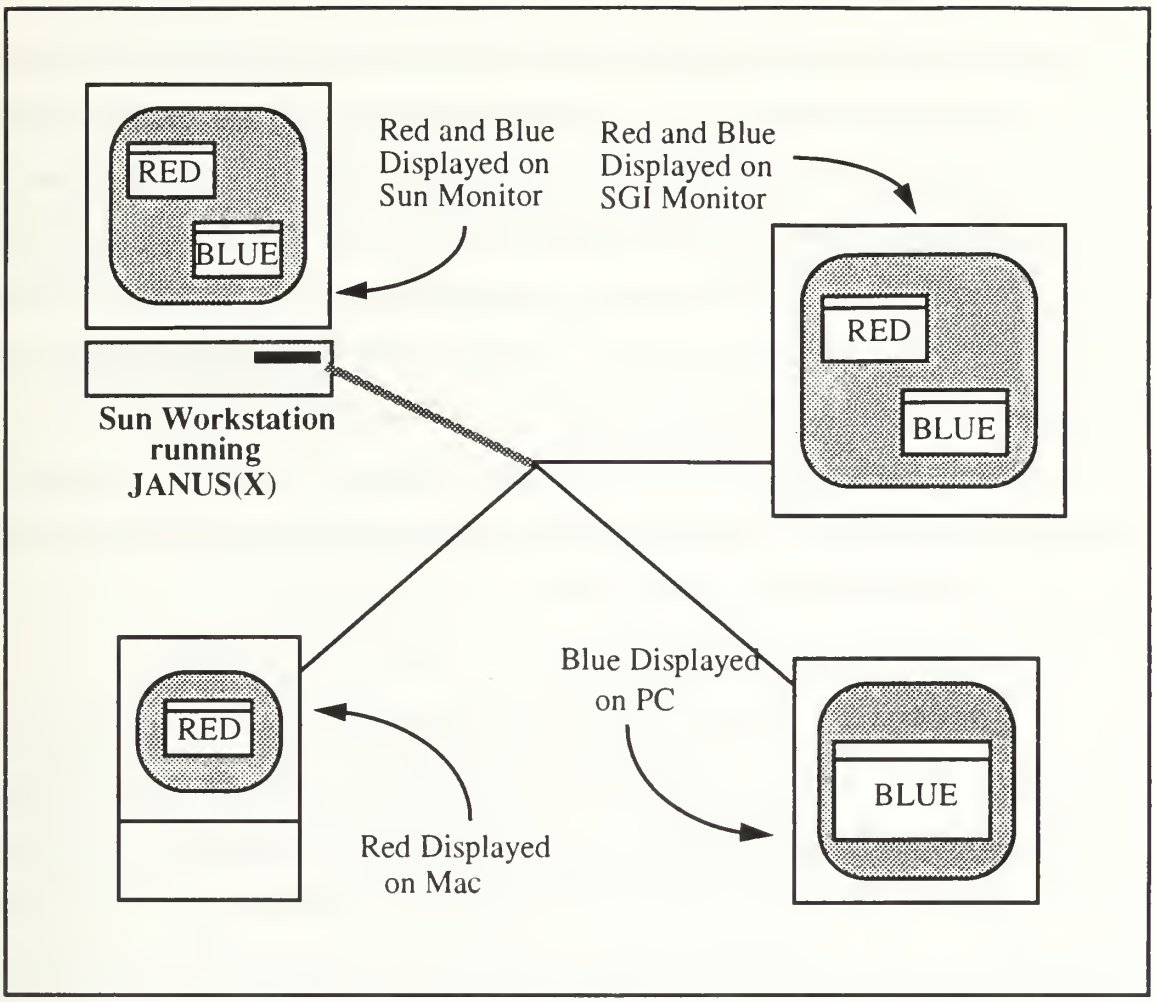

Figure 21: Networking Capabilities of JANUS(X)

The communications between the Sun Workstation, running the JANUS(A) simulation, and the workstations rendering the two-dimensional images is accomplished using an Ethernet network. JANUS(X) creates messages and places them on the network as packets. These message packets are read by a listening workstation and rendered appropriately on its monitor [GUYT 92].

JANUS(X) executes two different programs at the same time. The main JANUS(X) program operates on one Sun workstation. This program executes all operations of the JANUS(A) combat model, initializes the network, and sends the message packets to the network. The messages are prepared using the functions found in the sub-

program *ct.c*, located in the *janus/sun/Xgraf* directory. The second program is called *xtest*, located in the *janus/sun* directory. This program runs on the workstation that will render the X-Windows screens. *Xtest* captures the message packets that are on the network, then sends them to the sub-program *parse*. Here, the messages are returned to their original function formats, which are defined in the sub-program *ct.c* (Note: There are two programs named *ct.c*. The one in the JANUS(X) directory, *janus/sun/Xgraf*, creates messages. The other one, in the second workstation's directory, *janus/sun*, reads the messages.). The functions in *janus/sun/ct.c* call the X-Windows library functions required to render the two-dimensional images on the monitor. Additionally, screen inputs to JANUS (e.g. mouse picks) are captured by *xtest*, recorded into a message packet, and sent back to the machine running the main JANUS model [GUYT 92].

4. Network Message Format

NPSNET and JANUS(X) use two different message sending protocols, and consequently, use different message formats. To accommodate both functions, messages are first sent from the JANUS(X) game using its own format. When these messages are read by the program *xtest.c* on the receiving IRIS workstation, they are changed to the NPSNET message format. Then, they are transmitted to the program *network.c* on the receiving IRIS workstation.

a. JANUS(X) Message Format

The program *Xgraf/ct.c* has the important task of preparing packets to send across the network. Each of these packets contain, among other things, the specific graphics function calls, which are used to display the battle on a 'listening' monitor. Each of these message producing procedures has two common elements, besides the function itself. First, each screen (red force and blue force) has a unique file pointer, referred to as a "screen pointer", assigned to it. The variable containing the address of the screen pointer is labeled *gr_out* as shown in Figure 22. Additionally, JANUS(X) stores the number of the screen currently being addressed in the global variable *ramtek_*[0], which is either the integer

"one" or "two" referring to the blue or red screen respectively. Secondly, every object displayed on the screen is assigned a unique identifier. This identifier, an integer value, is referred to as the *segment_number* as depicted in Figure 22 [GUYT 92]. This *segment_number*, which is key in rendering the appropriate item on the monitor, can be anything from a tactical vehicle to a line that makes up a border of the screen. Another piece of important information, often included in a message, is the grid coordinate for the object referenced in the message.

When the JANUS(X) program calls one of the functions in the subprogram *Xgraf/ct.c*, the name of the function, along with a list of arguments representing appropriate drawing commands, are passed to the function *ct_printf* as illustrated in Figure 22. When called, the *ct_printf* function places the set of arguments from the message into a queue named *va_args*. When this argument queue reaches a total of 80 arguments, the queue is packaged into *one* message packet and placed on the network. As mentioned above, the 80 argument message packet is received by *xtest,* then parsed into the original set of function calls. All integer and real numbers are passed by reference with in the JANUS(X) program. However, these same integer and real numbers are passed by value within the message packets.

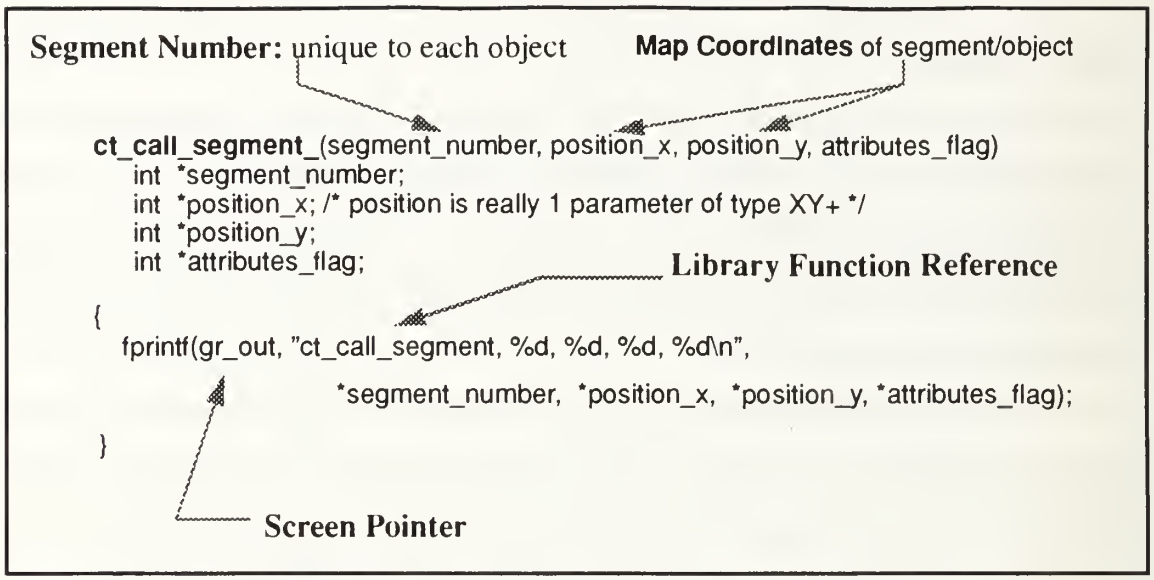

Figure 22: JANUS(X) Network Message Example

b. NPSNET Message Format

The NPSNET network protocol is simpler than that used by JANUS(X). Messages consist of a lead character, followed by a set of integer and real number arguments as shown in Figure 23. The lead character is used in the parsing process to indicate to which message function the incoming message belongs. The definition of the lead characters of the messages are defined in the file *network.h* of the NPSNET program. Instead of batch messages, NPSNET packages each message individually, using the 'C' *sprintf* function. The message is then placed on the network using the function *putmessonnet,* found in the program *janus/sun/ct.c.*

The NPSNET sub-program *network* parses all incoming messages using the lead character of the message as the key. Once parsed, the message is passed to the appropriate function in NPSNET.

```
ct_send_veh_fire_(numunit, numside, tgtunit, tgtside)
    int *numunit;
    int *numside;
    int *tgtunit;
    int *tgtside;
{
 char message[MESSAGELENGTH];
 sprintf(message,"%c %d %d %d %d"
      JANFIREMESS, *numunit, *numside, *tgtunit, *tgtside);
 putmessonnet(message);
}
```

Lead Character

B. NPSNET MESSAGES FROM JANUS

1. General Message Translation to NPSNET Format

a. Segment Number Conversion

The *segment_number* uniquely identifies every object that JANUS(X) wants drawn. The blue force vehicles are assigned the segment numbers between 600 and 1199, and the red force vehicles are assigned segment numbers from 1200 to 1799. The use of this "single-number" identifier presents a problem, because the remainder of the JANUS(X) model uses a "two-number" identifier (*force side* and *vehicle number*) to identify a vehicle. However, because the starting index for each force's vehicles is known, as well as the length of the segment of numbers assigned to the blue and red vehicles, it is possible to calculate the *force side index* and the *vehicle number*. To calculate the *force side* index, divide the segment_number by 600 using integer division as described in Equation 5.1. To calculate the vehicle *unit number* index, take the modulus of the *segment_number* base 600 as shown in Equation 5.2.

$$\text{Side} = (\text{Segment\_Number}) \; div \; (600) \qquad \text{(Eq 5.1)}$$

$$\text{UnitNumber} = (\text{Segment\_Number}) \bmod (600) \qquad \text{(Eq 5.2)}$$

b. Grid Coordinate Translation

Recall that JANUS(X)'s origin for its coordinate system is situated at the lower-left hand corner of the map. Also, the origin is not labeled (0, 0), but is labeled

according to the UTM coordinates assigned to this point in the world, in kilometers. NPSNET, on the other hand, has its origin in the upper-left hand corner of the map, and references this location by the coordinates (0, 0). Also, NPSNET measures distance in meters not kilometers. Therefore, for NPSNET to recognize the JANUS(X) coordinates these coordinates must be translated to the NPSNET system format. To further complicate matters, JANUS(X) masks the thousands digit of each coordinate in order to save room in memory. This digit must be reconstructed for NPSNET's use.

In the JANUS(A) terrain file, the map origin's UTM coordinates (coordinates of the lower-left hand corner of the map) are specified in their entirety, thus providing the information needed to restore the thousands digit. Dividing each UTM origin coordinate by 1000, using integer division, reconstructs the value of the thousands digit. Multiplying this value by 1000 yields a value which, when added to the JANUS x and z coordinates, provides the entire UTM coordinate (see Equation 5.3 and Equation 5.4). Once the thousands digit is restored to each of the JANUS grid coordinates, these coordinates are transposed to the NPSNET system using Equation 5.5 and Equation 5.6.

$$X_{\text{JANUS\_UTM}} = [\,(\text{X\_Origin}_{JANUS}/1000) + 1000] + X_{JANUS} \qquad \text{(Eq 5.3)}$$

$$Y_{\text{JANUS\_UTM}} = [\,(\text{Y\_Origin}_{JANUS}/1000) + 1000] + X_{JANUS} \qquad \text{(Eq 5.4)}$$

$$X_{NPSNET} = (X_{\text{JANUS\_UTM}} - \text{X\_Origin}_{JANUS}) \times 1000 \qquad \text{(Eq 5.5)}$$

$$Z_{NPSNET} = \text{Height\_of\_Map} - [\,(Y_{\text{JANUS\_UTM}} - \text{Y\_Origin}_{JANUS}) \times 1000] \qquad \text{(Eq 5.6)}$$

c. Vehicle Model Identification

Once a segment number is identified as belonging to a vehicle, its vehicle-type or system-type is determined (i.e. tank, truck, etc.). JANUS(A) references the attributes, which includes the system-type pertaining to a particular vehicle using a two tiered system. The first tier refers to a data structure called *systype*. *Systype* is a two-dimensional array that contains the name and other characteristics of a particular system (JANUS refers to a weapon or vehicle as a "system"). The first index into the *systype* array is the *force side* of a particular type of vehicle (e.g. M1A1 tank may have the force side

index "one", while a BMP may have the force side index of "two"). The second index, referred to as *numtype,* is a value between 0 and 49 inclusive, that selects the specific vehicle type in question. (e.g. A tank may have the index of five and a fuel truck the index of nine.). Once chosen, these systems characteristics are stored in the *systype* data structure.

The second tier in the vehicle identification process refers to a structure named *unitsdata. Unitsdata* is another global, two-dimensional array. This array uses the *force side* number of a vehicle and its *vehicle identification* number as indices. This array contains the *numtype* for each vehicle, which is needed to index into the *systype* array and ultimately identifying the vehicle's system-type. Figure 23 illustrates the indexing technique required to identify a vehicle's system-type.

```
numtype = unitdata[force_side][vehicle_number]
system_type = sysdata[force_side][numtype]
vehicle_type = system_type
```

Figure 23: Vehicle Type Identification

d. Elimination of Multiple Update Commands

JANUS(X) creates two separate screen images -- one for the blue force and one for the red force. Initially, vehicles that belong to each force are only rendered on the screen assigned to the owning force. However, as a vehicle of an opposing force arrives at a position on the map that enables it to observe some of the opposite force's vehicles, then these observed vehicles are rendered on both screens. This means that two identical graphics messages are sent across the network, indicating that the same image be rendered once on each screen. If the identical update messages are sent twice, NPSNET will try to update the image twice. Therefore, the side of the vehicle is checked against the number of the screen it is assigned. If the side and screen number do not match, then that update message is not passed to NPSNET as illustrated in Figure 24.

```
ct_delete_segment_(segment_number)
   int *segment_number;
{
   int *numunit, *numside, *screen;
   float *vehview;
   float xaddedamount, zaddedamount;
   double *xu, *yu; /* x and y coordinates */

   /* blue units go from 600 to 1199 and red units go from 1200 to 1799 */
   *numunit = *segment_number%600;

   /* blue is 1 red is 2 */
   *numside = *segment_number/600;

   /* displaying on red or blue screen? */
   *screen = ramtek_[0];

   /* only kill a vehicle if it is killed on its own screen */
   if((*numside == 1 && *screen == 2) || (*numside == 2 && *screen == 1)) {

        xaddedamount = (float)((int)MAPXORIGIN/1000) * KM2M;
        zaddedamount = (float)((int)MAPZORIGIN/1000) * KM2M;

        /* get x and y values and convert to npsnet values */
        /* unitsdata_ is a fortran common block */
        *xu = unitsdata_.xunit[*numside-1][*numunit-1];
        *xu = (*xu + xaddedamount - MAPXORIGIN) * KM2M;

        *yu = unitsdata_.yunit[*numside-1][*numunit-1];
        *yu = (*yu + zaddedamount - MAPZORIGIN) * KM2M;
        *yu = (ZTALL*KM2M) - *yu;

        /* get view direction and convert to npsnet values */
        *vehview = direct_.dview[*numside-1][*numunit-1];
        *vehview = TWOPI - *vehview;

        /* send kill message to npsnet */
        fprintf(gr_out, "ct_get_veh_kill_mess, %d, %d, %f, %f, %f\n",
                        *numside, *numunit, *xu, *yu, *vehview);
    }

    fprintf(gr_out,"ct_delete_segment, %d\n",*segment_number);
}
```

Reconstruct Side and Unit Indices

Check For Duplicate Message

Reconstruct Thousands Digit

Transpose Coordinates to NPSNET Coordinates

NPSNET Message

JANUS(X) Message

Figure 24: Sample Network Message Code

2. NPSNET Initialization

In order for NPSNET to read a JANUS(X) message and identify the specified vehicle in the NPSNET world, the vehicle array in the NPSNET system must be initialized as described in Chapter IV. To complicate this initialization process, JANUS(X) does not send any special initialization messages across the network. Therefore, there is not a one-

to-one message correspondence capable of providing the information needed to initialize NPSNET's vehicle array. However, two observations concerning JANUS(X) provide the capability to accomplish this initialization. The first being that JANUS(X) begins game play by drawing all vehicles to the screen once. The second is that there is one unique message that JANUS(X) sends just as game play commences, called *x_enable_gin*. Using these facts, the initialization process is started by sending NPSNET the message *ct_get_init_veh_mess* every time JANUS calls its own vehicle update message function, *ct_set_segment_position*. This initialization message is further flagged by a condition clause that checks to see if JANUS has sent the *x_enable_gin* message. If *x_enable_gin* was sent, then the boolean value, *start_boolean*, is set to true causing the initialization messages to be replaced with vehicle update messages.

Upon receipt of an initialization message, NPSNET assigns the incoming vehicle an NPSNET identification number referred to as *vehnum*. The identification number is then stored in a two-dimensional array called *janus2npsveharray*[side][unit_number], where the indices into the array are the JANUS *force side* and *unit number* of a vehicle. This new array is used to cross reference between the JANUS indices for a vehicle and the NPSNET index for a vehicle as shown in Figure 25. The initialization message contains two other important pieces of information. First the vehicle type, referred to as *iconnumber* in the code, is sent. NPSNET uses the *iconnumber* to lookup the NPSOFF model type. The table named *janicontable* contains the NPSNET model numbers. This model number, is then assigned as the vehicle type or *vehtype* of the vehicle. The other important piece of information sent in the message is the ***x*** and ***z*** coordinates of the vehicle. These values are recorded in the *vehpostype* structure assigned to the vehicle. (The vehpostype structure was discussed in Chapter IV.)

Upon initialization, all vehicles are set "alive" with a speed of zero. The control of the vehicle is set to SAF, indicating that the vehicle will receive all of its updates from the network. The ***y*** coordinate (elevation above sea level) for the vehicle is set to zero. NPSNET will calculate the vehicles true elevation using a function called *ground_level*

during execution of the graphics loop. If the NPSNET function, *isaircraft*, determines that the vehicle is an aircraft then the vehicle's elevation above the terrain is calculated using the *movetheaircraft* functions.

```
getjanusinitmess(message)
  char message[]; /*incoming message*/
{
 struct vehpostype tempveh;
 int iconnumber; /*the Janus icon number */
 int janside;   /*the side of the Janus vehicle */
 int janvehnum;  /*the vehicle unit number for Janus */

 if(networkvehiclesin == FALSE) networkvehiclesin = TRUE;
 /* Increment the total number of vehicles read in for JANUS in NPSNET */
     jannumvehs++;
                                                    Incoming Message Format
     sscanf(message, "%d %d %f %f %f %d",
        &janside, &janvehnum, &tempveh.pos[X],&tempveh.pos[Z],
        &tempveh.direction, &iconnumber);

   /* Set the new vehicle's index in the lookup table number */   Load JANUS to NPSNET
     jan2npsveharray[janside][janvehnum] = jannumvehs;          Cross Index Array

   /* initialize the NPSNET vehtype values */
     tempveh.vehtype = janicontable[iconnumber];
     tempveh.jan_type = iconnumber;
     tempveh.viewdirection = 0.0;
     tempveh.speed = 0.0;
     tempveh.alive = 1;
     tempveh.pos[Y] = 0.0;
     tempveh.behavoir = SAF;
     tempveh.undriven_control = SAF;

     if (canshoot(tempveh.vehtype)) tempveh.rounds = 100;
     else tempveh.rounds = 0;

     /* If the vehicle is an aircraft place it 100 meters above the ground */
     if(isaircraft(tempveh.vehtype))
           tempveh.elev = 100;
     else tempveh.elev = 0.0;

     /* Place the new vehicle in the vehicle array */
     veharray[jannumvehs] = tempveh;
     veharray[jannumvehs].vehnum = jannumvehs;
}
```

Figure 25: NPSNET Initialization Message Function

3. Vehicle Movement Updates

The JANUS(X) message *ct_set_segment_postion* is used to update the movement of a vehicle on one of the two-dimensional screens. Therefore, in the body of the JANUS(X) function to send this message, a message destined for NPSNET called

ct_get_veh_move_mess is embedded. This message includes the side and unit number of the vehicle, its speed, orientation and current coordinates. Using the segment_number provided by the JANUS(X) simulation, the speed and orientation (labeled *vehview*) of the vehicle are retrieved from the global structures *unitsdata.speed*[side][vehicle_number] and *direct.dview*[side][vehicle_number] respectively. Using the side and vehicle number as indices, NPSNET can adjust the location, speed and direction of the vehicle in the three-dimensional world.

4. Vehicle Shots

Just as with initialization of the vehicle set, JANUS(X) does not send any message which specifically identifies that a vehicle fired a shot. Therefore, a function was written to write this message to NPSNET, called *ct_send_veh_fire*. This message sends the force side and vehicle identification number for both the shooter and the intended target. Upon receipt of this message, NPSNET will render a muzzle flash for the vehicle that fired. Also, a red or blue line (depending on the force side of the firer) is drawn from the shooter to the target. This line simulates the tracer action of a shot, while the color allows the observer a quick reference to who fired it. The *ct_send_veh_fire* message function is called by the FORTRAN function *drawfire*. Every time the JANUS(X) sub-program, *intact.for*, indicates that a vehicle shot its weapon it calls the *drawfire* function.

5. Artillery

Just as with vehicle shots, JANUS(X) does not send any message which specifically identifies that artillery was fired. So again, a function was written to send an artillery message to NPSNET. The name of this message is *ct_send_arty*, and it sends the ***x*** and ***y*** coordinates where the artillery is currently landing. When this message is received by NPSNET, an artillery explosion is rendered at the coordinates indicated. The artillery message is initiated by the FORTRAN function *drawarty*. The JANUS(X) sub-program, *indfir.for*, upon indication of an artillery shot, calls the *drawarty* function.

6. Vehicle Destruction

The death or destruction of a weapon system in the JANUS(X) combat model is simulated by simply removing its icon from the two-dimensional screen. This removal is executed using the message *ct_delete_segment.* To notify NPSNET of the destruction of a vehicle, a message to NPSNET was embedded in the function *ct_delete_segment.* This new message sends the force side, vehicle identification number, grid coordinates and the final orientation of the vehicle. NPSNET then simulates the destruction of a vehicle by halting it and displaying an orange flame.

VII. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

1. Results

Two important results were achieved with this work. First, NPSNET was validated as a flexible, three-dimensional simulation platform for integration with traditional two-dimensional models. Second, it was shown that a traditional two-dimensional combat model can be successfully displayed in three-dimensions. This provides new life to systems that would otherwise rapidly become obsolete. What makes the second result even more noteworthy is that the three-dimensional display is achieved at a very low cost.

2. Applications

Currently, NPSNET provides the JANUS(A) combat modelers a method to verify many of their algorithms. For instance, visibility testing algorithms can easily be verified by moving a vehicle to a prescribed position on the battlefield and actually analyzing its true field of view. Another use of NPSNET:JANUS-3D is to provide the war-fighter operator immediate, realistic, visual feedback of an ongoing battle. This provides a more realistic training tool for the operator, as opposed to merely reading the synthesized data printed to the two-dimensional JANUS(A) screen. Also, NPSNET:JANUS-3D provides the combat weapons system developer the ability to easily visualize a new weapon system. The developer can develop appropriate tactics for the system, and test different weapon system specifications such as size, shape and positioning of optical devices prior to constructing any portion of the system. NPSNET: JANUS-3D is an excellent After-Action Review (AAR) tool. With this tool, the instructor possesses the capability of riding through the replay of an actual battle and providing salient observations of a vehicle's explicit conduct during an operation -- stopping and starting the replay as required.

B. RECOMMENDATIONS

There are several features that, if added to NPSNET:JANUS-3D, would further enhance the realism. First, there is a need to add temporary cultural objects such as mine fields and abatis. Second, when operating the simulation in a networked/real-time mode, the synchronization between the clock-speed of NPSNET and JANUS(A) is often misaligned. Because of this lack of synchronization, vehicles appear to jump on the screen. Therefore, a method to synchronize the clocks of the two models is desirable.

While this project proved that two-dimensional combat models can be displayed in three-dimensions, it only allows the user to be an "observer" of the simulation. The next logical step is to allow the observer to become an "operator" within the simulation. The goal is to have the actions of the operator within the three-dimensional world, be used as input to the two-dimensional model. This feature would allow war-fighters and combat model analysts the capability to test new ideas quickly and conduct multiple "what if" operations using the model.

The version of NPSNET used for this project used NPS networking formats, however, there is a working version of NPSNET that utilizes SIMNET networking protocols. A DIS version is also planned. When work is complete, the NPSNET:JANUS-3D connection will be the prototype for connecting DoD combat models to the DoD Defense Simulation Internet. Likewise, when "two-way" communication between the three-dimensional simulation, NPSNET, and the two-dimensional combat model, JANUS(A), is established, NPSNET will become the important interface between other three-dimensional combat simulations, such as SIMNET, and the traditional, two-dimensional combat models.

APPENDIX A: FILE TRANSLATION TECHNIQUES

A. FILE TRANSLATION FROM VAX TO IEEE

One of the first difficulties encountered when attempting to implement JANUS data files in NPSNET was a serious compatibility problem between the two systems. NPSNET, written in the C programming language, is run on the Silicon Graphics IRIS computers which uses the UNIX operating system. JANUS(A), on the other hand, consists entirely of code written in VAX-11 FORTRAN, a structured Digital Equipment Corporation (DEC) extension of standard FORTRAN-77, and operates on the VAX family of computers utilizing the VMS operating system [JANU 86].

Little-endian machines, such as the VAX, store words with the high-numbered byte as the most significant, while big-endian machines, such as the IRIS, have the low-ordered bytes of a word as the most significant.

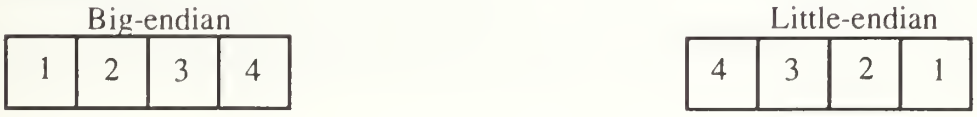

Figure 26: Big-Endian vs. Little-Endian Architectures.

In addition to the byte ordering problems of the two different machines, there is also a difference in the way floating point numbers are stored. The VAX architecture defines four floating-point formats. The only one that the JANUS data files utilize is the four-byte format. There also are two 8-byte formats, D_Floating and G_Floating, and one 16-byte format, H_Floating. As stated above, the VAX architecture is little-endian, which means that bits <7:0> are stored in the first byte, bits <15:8> in the second, bits <23:16> in the third, and <31:24> in the fourth.

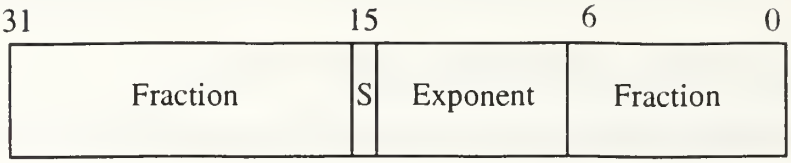

Figure 27: VAX Architecture, Four-Byte, Floating Point Format.

This is a sign-magnitude representation, with the sign in bit <15>, an excess 128 binary exponent in bits <14:7>, and a normalized 24-bit fraction in bits <6:0> and <31:16>. An exponent value of 0 and a sign bit of 0 are taken to indicate true zero, while an exponent value of 0 and a sign bit of 1 indicates a reserved operand (results in a reserved operand fault when processed). Exponent values of 1 through 255 indicate true binary exponents of -127 through +127 [VAX 90].

This known, the conversion from VAX to SGI for integers simply involves a reversal in the order of the bytes. The conversion for floats is slightly more involved. First, Byte 1 is exchanged with Byte 2. Then, Byte 3 has one subtracted, and is exchanged with Byte 4 [BUHL 91]. An example of the C code to accomplish the conversion for floats follows:

```
This is function vax_real_to_sgi_float. It accepts a vax
formatted float, (as an unsigned long in hex), and returns
the float value in IEEE format.


/* prototype*/
float vax_real_to_sgi_float(unsigned long);


float vax_real_to_sgi_float(unsigned long vax_real)
{
union VAXSTUFF
{
unsigned char byte[4]; /* array of four bytes used for */
float ieee_float; /* this is the returned ieee format */
unsigned long vax_int; /* holds the vms input format */
};

union VAXSTUFF indata;
unsigned char temp;


indata.vax_int = vax_real;

/* start swapping bytes */
if (indata.vax_int!= 00000000)
{
temp = indata.byte[1] - 1;
}

else
{
/* a 0.0 is a 0.0 in any format */
temp = indata.byte[1];
}

indata.byte[1] = indata.byte[0];
indata.byte[0] = temp;

temp = indata.byte[3];
indata.byte[3] = indata.byte[2];
indata.byte[2] = temp;

/* this is your answer */
return (indata.ieee_float);
}
```

Figure 28: Example Code for VAX to SGI Float Conversion

APPENDIX B: USER'S GUIDE

A. INTRODUCTION

This appendix provides a step-by-step description of the programs and procedures that must be performed to create the three-dimensional NPSNET:JANUS-3D virtual world and run the simulation in scripted and real-time modes. This appendix is divided into six sections:

- VAX to UNIX file translation
- Generation of three-dimensional terrain
- Generation of a script
- Operation of the two-dimensional replay simulation
- Operation of NPSNET with a script
- Operation of NPSNET and JANUS(X) in a networked, real-time mode

Within this document, all files and directories are written in *italics,* and all commands that must be entered at the key board are written in **`bold`**.

B. VAX TO UNIX FILE TRANSLATION

The following section provides a description of the programs and procedures that translate the JANUS binary, terrain, system, force, deploy, and post-processor files from a VAX to Unix format. Before starting, the user must ensure that the files listed above, are copied from the VAX/VMS system to a nine inch tape using the VMS **`copy`** command. These files are then loaded into the Unix system using the Unix **`vmstp`** command. These files must be copied in this exact manner, because the translation programs were written to accommodate the administrative bytes inserted by the VAX computer during the copy process (i.e. the block and record delineation bytes.).

To begin the file translation process, copy the following JANUS(A) files from the nine inch tape to the */n/gravy2/work/trac/janus/2dmapstuff/files* directory:

- terain###.dat
- system###.dat

- force###.dat
- dploy###.dat
- ppmove###.dat
- ppfirs###.dat
- ppmins###.dat
- pparty###.dat
- ppkils###.dat
- ppdtec###.dat

1. Translation of Terrain File

First, run the program *readtrrn*. It is located in the */n/gravy2/work/trac/janus/2dmapstuff/terrain* directory. This program reads the JANUS(A) file, *terain###.dat*, from the */n/gravy2/work/trac/janus/2dmapstuff/files* directory, and converts the data contained in the file to a Unix format. The converted data is written to the ASCII files *###.ele* (terrain elevation data), *###.riv* (river location data), and *###.road* (road location data), which are located in the */n/gravy2/work/trac/janus/2dmapstuff/files* directory. Additionally, the header file, *janus.h,* is created and written to the */n/gravy1/work2/pratt/simnet/coll2/headers* directory. The *janus.h* file contains all of the static definitions for the terrain database such as the size of the map, UTM coordinates for the lower-left hand corner of the map and the resolution value for each grid square. All other programs -- to include NPSNET -- use the definitions found in *janus.h* during execution.

Usage: **`readtrrn ###`**, where ### is the three number identifier from the terrain file *terrain###.dat*.

2. Recompilation

Once the terrain file is translated and *janus.h* is written, all programs pertaining to NPSNET:JANUS-3D must be recompiled. Therefore, the user must go to the directories listed in Figure 29 and type the following sequence of commands:

```
> rm *.o
> make
```

/n/gravy2/work/trac/janus/2dmapstuff/files
/n/gravy2/work/trac/janus/2dmapstuff/force
/n/gravy2/work/trac/janus/2dmapstuff/system
/n/gravy2/work/trac/janus/2dmapstuff/deploy
/n/gravy2/work/trac/janus/2dmapstuff/script
/n/gravy2/work/trac/janus/dirt/makeground

Figure 29: List of Directories Requiring Recompilation

3. Translation of the System File

Next, the program *readsysfiles* is executed. It is located in the */n/gravy2/work/trac/janus/2dmapstuff/system* directory. This program reads the JANUS(A) file, *system###.dat* from the */n/gravy2/work/trac/janus/2dmapstuff/files* directory, converts the data to Unix format and produces the ASCII files *sys_force.dat* (JANUS(A) system and weapons information), *name.dat* (weapon to system number references), and *cloud.dat* (cloud ceiling, cloud coefficients, and wind speed). These files are also written to the */n/gravy2/work/trac/janus/2dmapstuff/files* directory.

Usage: **`readsysfiles ###`**, where '###' are the three numbers associated with the JANUS(A) system file, *system###.dat*.

4. Translation of the Force File

The program *readforce* is executed next. It is located in the */n/gravy2/work/trac/janus/2dmapstuff/force* directory. This program reads the JANUS(A) file, *force###.dat* from the */n/gravy2/work/trac/janus/2dmapstuff/files* directory. It converts the data to Unix format and produces three ASCII files called *sysnames.dat* (matches weapon to its system), *force.data* (matches a vehicle to its side and system), and *icontable.dat* (matches a two-dimensional icon to a vehicle). These files are written to the */n/gravy2/work/trac/janus/2dmapstuff/files* directory.

Usage: **`readforce ###`**, where '###' are the three numbers associated with the JANUS(A) force file, *force###.dat*.

5. Translation of the Deployment File

The program *readdeploy* is executed next. It is located in the */n/gravy2/work/trac/janus/2dmapstuff/deploy* directory. This program reads the JANUS(A) file, *dploy###.dat* from the */n/gravy2/work/trac/janus/2dmapstuff/files* directory. It converts the data to Unix format and produces the ASCII file *initunit.dat* (initial locations of vehicles). This file is written to the */n/gravy2/work/trac/janus/2dmapstuff/files* directory.

Usage: **`readdeploy ###`**, where'###' are the three numbers associated with the JANUS(A) deployment file, *dploy###.dat*.

6. Translation of Post-processor Files

The program *readppfiles.c* is executed after the terrain, force, system and deployment files are translated. It is located in the */n/gravy2/work/trac/janus/2dmapstuff/files* directory. This program reads the JANUS(A) files, *ppmove###.dat*, *ppfirs###.dat*, *ppmins###.dat*, *pparty###.dat*, *ppkils###.dat*, and *ppdtec###.dat*, which are located in the */n/gravy2/work/trac/janus/2dmapstuff/files* directory. It converts the data to Unix format and produces the ASCII files called *ppmove.dat*, *ppfire.dat*, *ppmines.dat*, *pparty.dat*, *ppkills.dat*, and *ppdtec.dat* respectively. These files are written to the */n/gravy2/work/trac/janus/2dmapstuff/files* directory.

Usage: **`readppfiles ###`**, where'###' are the three numbers associated with each JANUS(A) post-processor files.

C. TERRAIN GENERATION

All the programs used to generate the three-dimensional terrain are located in the */n/gravy2/work/trac/janus/dirt/makeground* directory. These programs must be generated in the exact order specified below. Additionally, generation of the polygonized terrain is both time consuming and memory consuming. Most files can be deleted once all of the terrain is generated. The names of the files that must remain are listed at the end of this section.

Before executing any programs, erase everything in the directories listed in Figure 30 using the Unix, **rm *** command.

/n/gravy2/work/trac/janus/dirt/coverfiles
/n/gravy2/work/trac/janus/dirt/textcoverfiles
/n/gravy2/work/trac/janus/dirt/elevfiles
/n/gravy2/work/trac/janus/dirt/textelevfiles
/n/gravy2/work/trac/janus/dirt/objectfiles
/n/gravy2/work/trac/janus/dirt/textobjectfiles
/n/gravy2/work/trac/janus/dirt/quadfiles
/n/gravy2/work/trac/janus/dirt/textquadfiles
/n/gravy2/work/trac/janus/dirt/roadriverfiles

Figure 30: Terrain Data Directories

Once these directories are clear then execute the following programs in the order listed:

(1) genbinaryelev: This program reads the file *###.ele* found in the */n/gravy2/work/trac/janus/2dmapstuff/files* directory, and creates the binary elevation file named *elev.bin.dat* in the */n/gravy2/work/trac/janus/dirt/elevfiles* directory.

Usage: **genbinaryelev ###**, where'###' are the three numbers associated with the file, *###.ele*.

(2) make_tri_mesh: This program reads the file *###.ele* found in the */n/gravy2/work/trac/janus/2dmapstuff/files* directory, and calculates the point normals for each elevation in the elevation file. It then writes this information to a binary file named *elev.mesh.dat* in the */n/gravy2/work/trac/janus/dirt/elevfiles* directory.

Usage: **make_tri_mesh ###**, where'###' are the three numbers associated with the file, *###.ele*.

(3) conv_elev2block_bin: This program reads the file *###.ele* found in the */n/gravy2/work/trac/janus/2dmapstuff/files* directory. Each elevation is then rewritten, in binary, to a file in the */n/gravy2/work/trac/janus/dirt/elevfiles* directory. The file that it is written to corresponds to the quadnode in which it is located. The generic name for these

new elevation files is *elevXXXXZZZZ.dat,* where XXXX and ZZZZ are the quadnode indices for the file.

Usage: **conv_elev2block_bin**

(4) janus2nps: This program reads the file *###.ele* found in the */n/gravy2/work/trac/janus/2dmapstuff/files* directory. This data is then converted to polygonized terrain data and converted to David R. Pratt (NPSDRP) data format [PRAT 92a]. The new data is then written in ASCII, to the file cover.dat, found in the */n/gravy2/work/trac/janus/dirt/textcoverfiles* directory.

Usage: **janus2nps ###,** where '###' are the three numbers associated with each file, *###.ele*.

(5) reverseroads: This program reads the files *###.road* and *###.riv* found in the */n/gravy2/work/trac/janus/2dmapstuff/files* directory. These files contain the coordinates for each road and river checkpoint. These coordinates are converted to the NPSNET coordinate system and rewritten to the ASCII files *road.dat* and *riv.dat*, respectively. Both of these files are located in the directory, */n/gravy2/work/trac/janus/dirt/roadrivfiles*.

Usage: **reverseroads ###**, where '###' are the three numbers associated with the files, *###.riv* and *###.road*.

(6) makeroadfile: This program reads the files *road.dat* and *riv.dat*, found in the */n/gravy2/work/trac/janus/dirt/roadriverfiles* directory, and creates two point line segments with the data. This new information is then written to the ASCII file *roads.dat* found in the */n/gravy2/work/trac/janus/dirt/roadrivfiles* directory.

Usage: **makeroadfile**

(7) makeroads: This program reads the file *roads.dat* found in the */n/gravy2/work/trac/janus/dirt/roadriverfiles* directory and converts the data into DRP polygonized terrain format. This new information is then written to the ASCII file *roadcover.dat* found in the */n/gravy2/work/trac/janus/dirt/roadrivfiles* directory.

Usage: **makeroads**

(8) maketrees: This program reads the file, *###.ele,* found in the */n/gravy2/work/trac/janus/2dmapstuff/files* directory and the file *roadcover.dat* found in the */n/gravy2/work/trac/janus/dirt/roadrivfiles* directory. It then generates the locations for tree and city canopies, and stores this information in the files treeblock.dat and cityblock.dat respectively. These two files are found in the */n/gravy2/work/trac/janus/dirt/textobjectfiles* directory. The program also calculates the locations for individual trees and buildings and sorts them by their height/density factors into the files *city1.dat* through *city7.dat* and *tree1.dat* through *tree7.dat.* These files are also written to the */n/gravy2/work/trac/janus/dirt/textobjectfiles* directory.

Usage: **maketrees**

(9) genblockcov: This program reads the file, cover.dat, from the directory, */n/gravy2/work/trac/janus/dirt/tectcoverfiles,* and the file, *roadcover.dat,* from the directory, */n/gravy2/work/trac/janus/dirt/roadrivfiles.* The elevation data from these files is rewritten, in ASCII, to a file in the */n/gravy2/work/trac/janus/dirt/textcoverfiles* directory. The generic name for these new elevation files is *coverXXXXZZZZtext.dat,* where XXXX and ZZZZ are the quadnode indices for the file. The file name, that the data is written to, corresponds to the quadnode in which it is located.

Usage: **genblockcov**

(10) conv_blockcov2bin: This program uses, as input, the files that were produced by the program *genblockcov.* The generic name for these files are *coverXXXXZZZZtext.dat.* They are found in the */n/gravy2/work/trac/janus/dirt/textcoverfiles* directory. The program rewrites the files to binary formatted files, found in the */n/gravy2/work/trac/janus/dirt/coverfiles* directory. The new generic name for the files is *coverXXXXZZZZbin.dat,* where XXXX and ZZZZ are the quadnode indices for the file.

Usage: **conv_blockcov2bin**

(11) genquadcov: This program reads the files, *coverXXXXZZZZbin.dat,* from the */n/gravy2/work/trac/janus/dirt/coverfiles* directory. In this program, the polygons for each resolution level (high, medium-high, medium-low and low) are created, then each file is converted into a quadtree format. Each file is then written to files with the generic

name of *coverXXXXZZZZtextquad.dat* found in the */n/gravy2/work/trac/janus/dirt/textquadfiles* directory.

Usage: **genquadcov**

(12) conv_quadcov2bin: This program uses, as input, the files that were produced by the program *genquadcov*. The generic name for these files are *coverXXXXZZZZtextquad.dat*. These files are found in the */n/gravy2/work/trac/janus/dirt/textquadfiles* directory. These files are rewritten to binary files found in the */n/gravy2/work/trac/janus/dirt/quadfiles* directory. The new generic name for the files is *coverXXXXZZZZbinquad.dat*, where XXXX and ZZZZ are the quadnode indices for the file.

Usage: **conv_quadcov2bin**

(13) genblockobj: This program reads the files *treeblock.dat*, *city block.dat*, *tree1.dat* through *tree7.dat*, and *city1.dat* through *city7.dat*, all from the */n/gravy2/work/trac/janus/dirt/textobject* directory. The data from these files is rewritten to ASCII files in the */n/gravy2/work/trac/janus/dirt/textobjectfiles* directory. The file that the data is written to, corresponds to the quadnode in which it is located. The generic name for these new elevation files is *objectXXXXZZZZtext.dat*, where XXXX and ZZZZ are the quadnode indices for the file.

Usage: **genblockobj**

(14) conv_block_obj_to_bin: This program uses, as input, the files that were produced by the program *genblockobj*. The generic name for these files are *objectXXXXZZZZtext.dat*. These files are found in the */n/gravy2/work/trac/janus/dirt/textobjectfiles* directory. The program rewrites these files to binary files and places them in the */n/gravy2/work/trac/janus/dirt/objectfiles* directory. The new generic name for the files is *objectXXXXZZZZbin.dat*, where XXXX and ZZZZ are the quadnode indices for the file.

Usage: **conv_block_obj_to_bin**

Once the above programs are completed, all files may be erased except the following:

- All files in the */n/gravy2/work/trac/janus/dirt/quadfiles* directory.

- All files in the */n/gravy2/work/trac/janus/dirt/objectfiles* directory.
- The file elev.bin.dat in the */n/gravy2/work/trac/janus/dirt/elevfiles* directory
- The file elev.mesh.dat in the */n/gravy2/work/trac/janus/dirt/elevfiles* directory

At this point, the data for the t-mesh and polygonized terrain is created. The user can now write the script, or run NPSNET in the real-time mode.

D. SCRIPT GENERATION

To generate a three-dimensional script, the following files must be present in the */n/gravy2/work/trac/janus/2dmapstuff/files* directory:

- ppmove.dat
- ppfires.dat
- ppkills.dat
- pparty.dat
- ppmines.dat

Before writing the script, the user must delete all files from the */n/gravy2/work/trac/janus/2dmapstuff/files/script_data* directory. These files must be removed because many of the script routines 'append' information to files, rather than 'create' new files. These files can be erased by moving to the */n/gravy2/work/trac/janus/2dmapstuff/files/script_data* directory and typing **rm *** at the prompt.

Once the directory is empty, the script is written by executing the two programs listed below:

(1) makeinitialposfile: This program is located in the */n/gravy2/work/trac/janus/dirt/makeground* directory. As input, it uses the files *initunit.dat* and *sysnames.dat*, both located in the */n/gravy2/work/trac/janus/2dmapstuff/files* directory. First, this program creates the file *janusvehiclepos.dat* and places it in the */n/gravy2/work/trac/janus/dirt/textobjectfiles* directory. This file contains the initial positions and view directions for each vehicle used in the script. Also the program creates a separate file for each vehicle in the */n/gravy2/work/trac/janus/2dmapstuff/files/script_data* directory. The generic name for these files is *newveh####.dat*, where the first # represents the integer value for the side of

the vehicle (one = blue, two = red), while the next three ###'s represent the vehicle's identification number.

Usage: **makeinitialposfile**

(2) writescript: This program is located in the */n/gravy2/work/trac/janus/2dmapstuff/script* directory. It uses the files *ppmove.dat, ppfires.dat, pparty.dat, ppkills.dat,* and *ppmines.dat*, as discussed above. This program writes a series of files named *veh####.dat*, and *realnewveh.dat* to the */n/gravy2/work/trac/janus/2dmapstuff/files/script_data* directory. These files are only used by this program and can be erased upon the conclusion of the program's execution. This program ultimately produces the file *janus_script.dat*, which is also written to the */n/gravy2/work/trac/janus/2dmapstuff/files/script_data* directory. This file contains the script that is used by NPSNET:

Usage: **writescript**

E. Operation of the Two-Dimensional REPLAY Program

The name of the program used to display the converted two-dimensional terrain, vegetation, and cultural features is 'REPLAY'. In addition to displaying the map, this program also tests the converted initialization and post-processor files by means of a scripting capability. 'REPLAY' is found in the */n/gravy2/work/trac/janus/2dmapstuff/terrain* directory.

1. Initialization and Start-Up Procedures

Before starting the program, all of the necessary files must be converted, and stored in the */n/gravy2/work/trac/janus/2dmapstuff/files* directory. These files include the terrain, initialization, and post-processor files. For a detailed description of the generation, names, and locations of these files (see "VAX TO UNIX FILE TRANSLATION" earlier in this Appendix).

To start the program, simply type **replay ###.** '###' is the three digit number corresponding to the two-dimensional digitized terrain map being used in the scenario.

2. Program Manipulation

The two major functions of the 'REPLAY' program are (1) the terrain feature display capability, and (2) the tactical scenario playback capability.

After starting the program, the first screen to appear is the grey-scale map as shown in Figure 1. As illustrated in Figure 31, clicking and holding the right mouse button, displays the **Main Menu**. To render the different terrain features, simply slide the cursor into the **Terrain Features Menu**, and select the desired feature. An example, after selecting cities, vegetation, roads, and rivers, is shown in Figure 2. By selecting an item from the menu once more, each terrain feature is removed from the display.

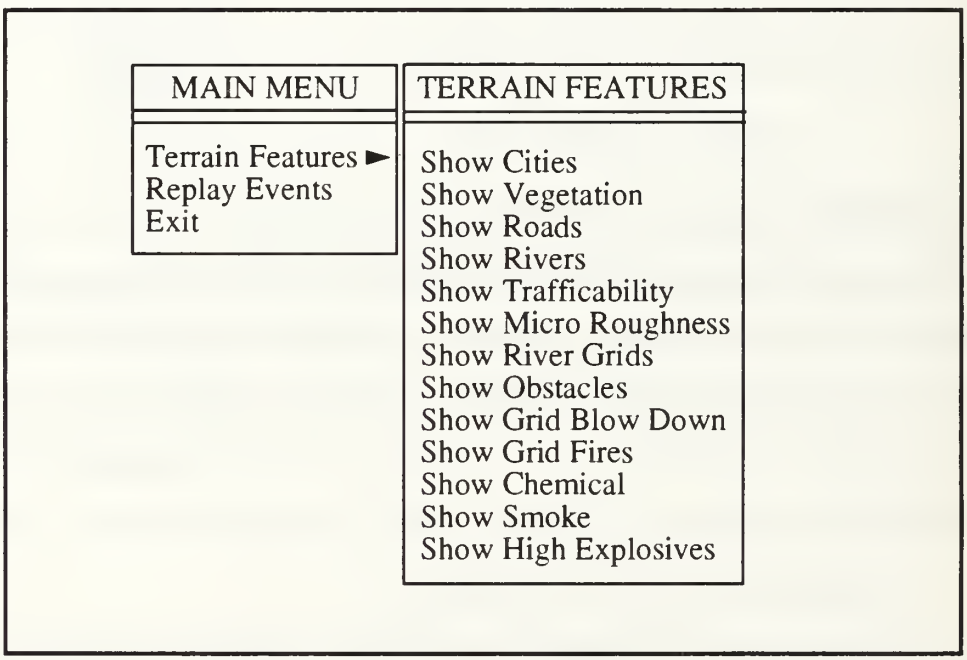

Figure 31: REPLAY Main Menu and Terrain Feature Sub-Menu

The second function of the 'REPLAY' program is the playback capability. To start the script, click on the **Replay Events** item of the **Main Menu**. After placing the Blue and Red force vehicles and all minefields in their initial positions, the script of the battle is immediately started.

An example of the playback program is shown in Figure 3. Movement events are shown by moving the two-dimensional icon to its next position. Direct fire events are drawn with straight lines -- blue lines represent friendly fire and red lines represent enemy fire. Indirect fire events are depicted as circles. The edges of the circles are colored according to the side (red or blue), and the interior of the circle is painted black for high explosive or white for smoke. Smoke rounds remain on the map until their determined dissipation time. Destroyed vehicles are changed to a green color, and remain on the map at that position.

Currently, the speed in which the script executes is equal to the recorded game time. Since some of the scenarios are over two hours in length, this requires the user to watch the entire battle, when the last 30 minutes may be the only interesting part. For future work, a nice feature to include would be the ability to speed up the game clock, in order to move quickly to a point of interest. A menu item has been included for this, which will increase the clock by a certain time factor (see Figure 32).

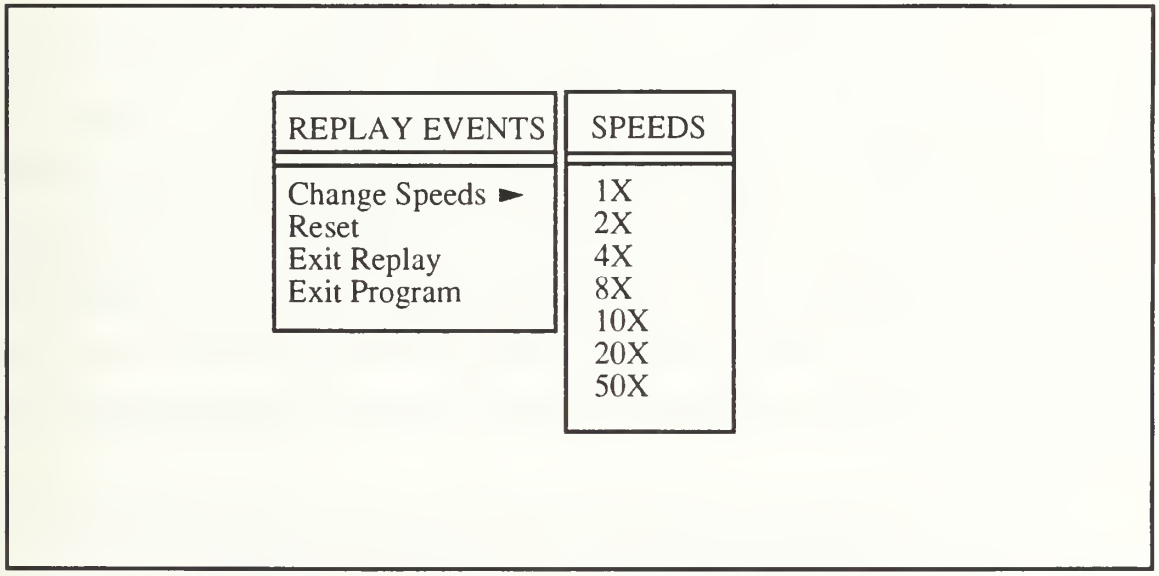

Figure 32: Replay Events Menu and Clock Speed Sub-Menu

After **Replay Events** is executed, the script can be reset to the beginning of the battle by selecting the **Reset** item. Likewise, the user can exit the scripting tool and return

to the **Main Menu** by selecting **Exit Replay**, or can exit the entire program by selecting **Exit Program** as shown in Figure 32.

F. Operation of NPSNET: JANUS-3D

There are currently two different methods of running NPSNET: JANUS-3D. The first is a Script Mode, and the second is a Real-Time Mode, which runs concurrently with JANUS(X) over a network. Different compilations are required for JANUS-3D, depending on which mode is chosen, and what terrain database is used. This section discusses the steps necessary to compile the different executable programs, followed by a detailed discussion of the operation of NPSNET: JANUS-3D in the two different modes.

1. NPSNET: JANUS-3D Compilation

Currently, NPSNET can be compiled in three different modes by using the 'C' Programming Language's **#ifdef** command during compilation. The three different variables presently used in the **#define** command are:

- NTC -- terrain and object data from the National Training Center
- HTL -- terrain and object data from Fort Hunter Liggett, CA
- JANUS -- terrain and object data generated from any JANUS(A) scenario

The option chosen decides which variable must be defined in the file */n/gravy1/work2/pratt/simnet/sdis/coll2/SYSTYPE.h*. Only the 'JANUS' option will be discussed, since the NTC and HTL versions of NSPNET are never used when running JANUS-3D. Besides ensuring that the *SYSTYPE.h* file is properly defined, there are two additional files that need adjusting in order to compile the proper version of JANUS-3D -- *janus.h* and *files.h*. Both of these files are found in the */n/gravy1/work2/pratt/simnet/sdis/coll2/headers* directory.

As discussed earlier, *janus.h* is created by the program *readtrrn*. The file janus.h contains the static definitions that pertain to a particular JANUS(A) database. When a new JANUS(A) database is translated for use with NPSNET, the *janus.h* file that is created for it is often stored under a temporary name for future use. The temporary name includes the

word 'janus', followed by a suffix (e.g. *janus_fulda*). The suffix is the name of the map used in the terrain database. Because the *janus.h* file is crucial in the creation of the proper three-dimensional terrain for NPSNET, it is important that the appropriate version is named "*janus.h*" during the compilation process.

The second file, *files.h*, contains all of the **#defines** for every file that is written to or read from in NPSNET. For JANUS-3D, several options are available in *files.h*, of which currently there are four: (1) Hunter Liggett Scripted Version, (2) Fulda, Germany Scripted Version, (3) Fulda, Germany Real-Time Version, and (4) 73 Easting Scripted Version. The #defines for these different options are separately grouped, with only one being readable (uncommented) at a time.

After *SYSTYPE.h, janus.h* and *files.h* are properly formatted, NPSNET can then be compiled and run. The following discussions of the operation of NPSNET: JANUS-3D are based on the assumption that the user is familiar with the operation of NPSNET, thus only the JANUS-3D peculiarities are discussed.

2. Operation of NPSNET: JANUS-3D in Script Mode

Assuming the proper version of the program was compiled, and the files are in the proper location, start JANUS-3D in the scripted mode by typing **npsnet** in the */n/gravy1/work2/pratt/simnet/sdis/coll2* directory (If operating on an SGI, Indigo-Elan type **npsnet t**.). When initialization is complete, and NPSNET is started, bring up the two-dimensional map by selecting the **2d Map Window** item of the **Main Menu** as shown in Figure 33. This will display the entire JANUS(A) grey-scale map, in the upper-right hand side of the screen, with the two-dimensional icon symbols in their initial positions. The viewer's location will be in the observer-controller vehicle. Before the scripting action is started, the user can place himself inside any vehicle by picking the appropriate icon in the 2d map window. Once inside, the vehicle's position and viewing angle can be changed with the Space Ball. Later, when the script is started, each of the vehicles that were moved will immediately return to there initial location and orientation.

To begin the script, select the **Play Tracks** item from the sub-menu, **Script Menu** as depicted in Figure 33. Like the two-dimensional REPLAY program, the speed of the script runs at game time, and cannot be increased. However, once started, the script can be reset by selecting the **Reset Tracks** item in the **Script Menu** as shown in Figure 33. While the script is in process, the viewer can take semi-control of any vehicle with the Space Ball. This control is limited, because the script file takes precedence over the user's inputs. In other words, the vehicle will jump back to its proper location and orientation every time the script tells it to.

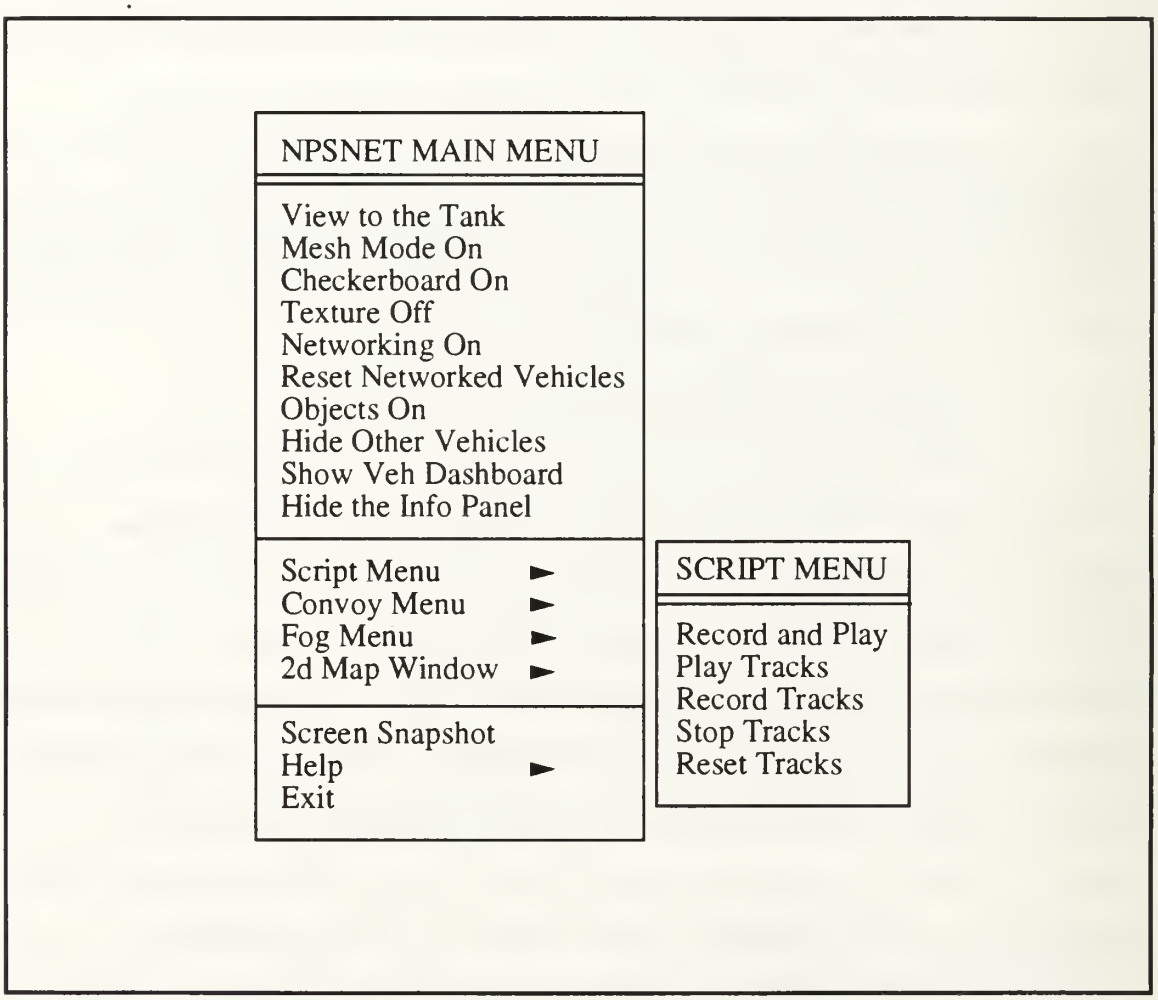

Figure 33: NPSNET Main Menu and Script Sub-Menu

3. Operation of NPSNET: JANUS-3D in Real-Time Mode

To run JANUS-3D in a real-time mode, two different programs must be started; (1) NPSNET, in the network mode, and (2) JANUS(X), from an IRIS Workstation.

NPSNET must be started, first, by typing **npsnet**. Once NPSNET is running, the networking mode must be turned on by selecting the **Networking On** item from the **Main Menu** as illustrated in Figure 33. After NPSNET is initialized and in the **Networking On** or 'listening mode', JANUS(X) can be started from another IRIS Workstation.

Before beginning JANUS(X), the user must be logged in on the workstation as TRAC. This is because the *.login* and *.cshrc* files in the TRAC directory contain aliases and path designations necessary to run JANUS(X).

To start, type the command **makejanwin**. The program, *makejanwin*, is a shell script, located in the */n/gravy2/work/trac/bin* directory and is listed in Figure 34.

```
wsh -f Screen14 -m 80,80 -s 60,14 -p 700,260 -n "RED SIDE"
wsh -f Screen14 -m 80,80 -s 60,14 -p 700,10 -n "BLUE SIDE"
xwsh -fn Screen14 -max 120x120 -geom 92x32+14+600 -title "JANUS DRIVER" -e rlogin libra
```

Figure 34: Makejanwin Shell Script

Makejanwin displays three separate windows at the bottom of the screen. The JANUS DRIVER screen at the lower-left corner is the main console window, from which JANUS(X) is started. The main console window is remotely logged into the **libra** file server, because JANUS(X) requires the FORTRAN run-time libraries that NPS stores on this file server. The other two windows are used to type the commands necessary to display the Red and Blue side screens, and are titled RED SIDE and BLUE SIDE, respectively.

In the RED SIDE window, type **red**. This will display the window for the red side, which can be positioned anywhere on the screen. Typing **blue** in the BLUE SIDE window, will duplicate the actions taken for the red window. These commands, **red** and

blue, are aliases which are defined in the *.login* file. The aliased commands, described in Figure 35, set each of the windows to a 'listening mode', assign a port number (to uniquely identify the window for packet sending), and begin the program *xtest*, which captures the message packets and calls the appropriate graphic functions.

```
alias blue 'setenv JANUS_LISTEN 20002;cd ~trac/janus/sun;xtest -x font.0'
alias red 'setenv JANUS_LISTEN 20001;cd ~trac/janus/sun;xtest -x font.0'
```

Figure 35: Example of Aliased Commands for JANUS(X) Initialization

The next step is to begin the JANUS(X) model in the JANUS DRIVER window. Type **xterm** at the TERM = (sun) prompt. Then, depending which IRIS Workstation is being used, type in the command **runjan_*irisname***, with ***irisname*** corresponding to the workstation. For example, if the model is being displayed on gravy1, type **runjan_gravy1**. These commands are also aliases which are defined in the 'trac' account's *.login* file. The aliased commands, as shown in Figure 36, produce two sending stations (one blue and one red), with the same unique port numbers as the listening windows. They also start the program, **janus**.

```
alias runjan_gravy1 'setenv JANUS_TERMINAL_1 gravy1:20001;
                     setenv JANUS_TERMINAL_2 gravy1:20002;
                     cd ~trac/janus/sun/janus/Programs;janus'
```

Figure 36: Aliased Command to Run JANUS on Gravy1 Terminal

After typing the **runjan** command, the JANUS DRIVER window will prompt the user for the Scenario Number and the Run Number of the specific JANUS(X) scenario. After entering the appropriate numbers, three JANUS(X) initialization screens will be presented in succession. Type in the necessary data into each screen followed by

<ENTER>. (NOTE: The <RETURN> key does not suffice, <ENTER> must be used). JANUS(X) will now be running in the two windows marked BLUE SIDE and RED SIDE.

Once the two screens are initialized, the JANUS(X) grey-scale map, in the upper-right hand side of the NPSNET screen, will render the two-dimensional icon symbols in their initial positions. As in the scripted mode, these vehicles can be entered by picking the appropriate icon from the 2d map. Once the real-time link is started, any altered vehicles are returned to their initial positions and orientations.

To start the real-time model, select the EXIT item from both the BLUE SIDE and RED SIDE windows as depicted in Figure 37. The JANUS DRIVER window will prompt the user to press <RETURN>. After pressing <RETURN>, the real-time version of JANUS-3D is started. While the real-time battle is in process, the viewer can take semi-control of any vehicle with the Space Ball, but loses control whenever a vehicle receives a message from the network.

To end the real-time simulation, select the ADMIN item from both the BLUE SIDE and RED SIDE windows as shown in Figure 37. The JANUS DRIVER window will display the Admin Menu. Selecting the **EJ** item (End Janus) will stop JANUS(X). When JANUS(X) is ended, NPSNET will continue updating vehicle positions according to their last speed and location. To clear the NPSNET screen, select the Reset Networked Vehicles item from the NPSNET Main Menu. Then, if needed, the JANUS(X) model can be restarted without having to restart NPSNET.

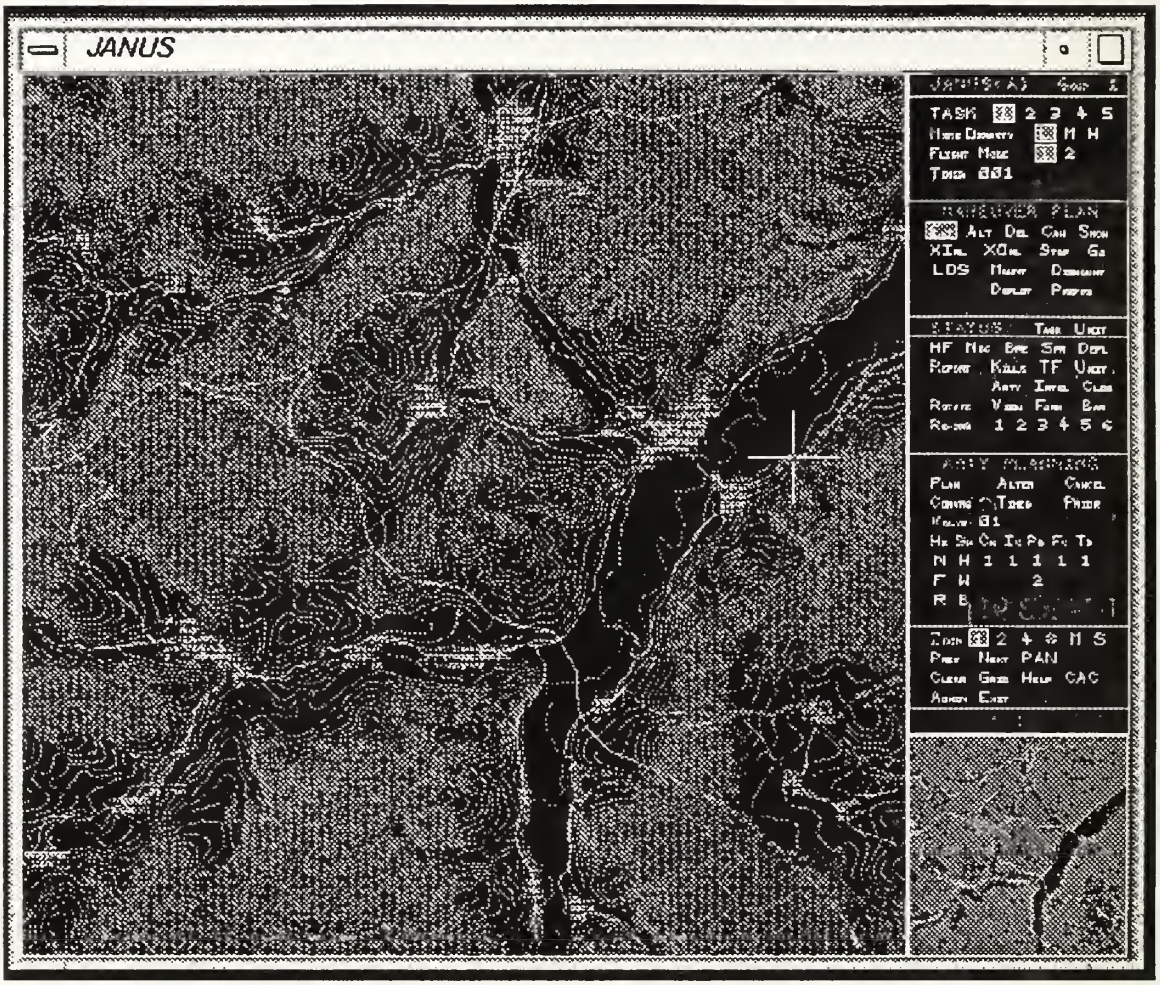

Figure 37: Example of the RED SIDE JANUS(X) Screen

LIST OF REFERENCES

[BUHL 91] Buhle, E. L., "Computing in a Heterogeneous Environment", *VAX-Profession*, pp. 11-13, August 1991.

[CLIP 91] Software Technology Branch, Lyndon B. Johnson Space Center, *CLIPS Reference Manual*, Version 5.0" JSC-22948, Jan 1991.

[GIAR 89] Giarratano, Joseph C., and Riley, Gary, *Expert Systems, Principles and Programming*, PWS-KENT, Publishing Company, 1989.

[GIAR 91] Giarratano, Joseph C., *CLIPS User's Guide*, Information Systems Directorate, Software Technology Branch, Lyndon B. Johnson Space Center, Jan 1991.

[GUYT 92] Guyton, Jim, *JANUS(A) for Sun/Unix*, Rand Corporation, Los Angeles, CA, April 1992.

[HARB 91] Harbison, Samuel P. and Steele, Guy L. Jr., *C, A Reference Manual*, 3rd Edition, Prentice Hall, 1991.

[JANU 86] U.S. Army TRADOC Analysis Command, WSMR, *JANUS(T) Documentation Manual*, June 1986.

[JANU 91] U.S. Army TRADOC Analysis Command, WSMR, *JANUS(A) Version 2.0 Information Letter*, March 1991.

[JANU 92] U.S. Army TRADOC Analysis Command, Leavenworth, *Janus Army, Single Model Serving All Domains*, Leavenworth, KS, 1992.

[KANA 91] Kanayama, Yutaka, "Advanced Robotics Course Notes", Naval Postgraduate School, Monterey, CA, Fall 1991.

[MACK 91] Macky, Randall L., *NPSNET: Hierarchical Data Structures for Real-Time Three-Dimensional Visual Simulations,* Master's Thesis, Naval Postgraduate School, Monterey, California, September 1991.

[PRAT 90] Pratt, David R., "showelev.c", computer program, 1991.

[PRAT 92] Pratt, David R., Zyda, Michael J., Mackey, Randall L., and Falby, John S., "NPSNET: A Networked Vehicle Simulation with Hierarchical Data Structures", Naval Postgraduate School, 1992.

[PRAT 92a] Pratt, David R. and Zyda, Michael J., "NPS Terrain Database Format", technical note generated by the Graphics and Video Laboratory for DARPA/ ANSTO and USATEC, 5 March 1992.

[SGI 91] Silicon Graphics Inc., *IRIS WorkSpace Integration Guide*, August 1991.

[SUN 91] Sun Microsystems INC, *Sun FORTRAN User's Guide*, February 1991.

[VAX 90] VAX MACRO, *Instruction Set Reference Manual (AA-LA89A-TE)*, pp. 8-3. 1990.

[ZYDA 91] Zyda, Michael J. and Pratt, David R., "NPSNET: A 3D Visual Simulator for Virtual World Exploration and Experimentation", *1991 Society for Information Display International Symposium Digest of Technical Papers*, Volume XXII, May 1991, pp 361-364.

[ZYDA 91a] Zyda, M. J., Wilson, K. P., Pratt, D. R. and Monahan, J. G., "NPSOFF: An Object Description Language for Supporting Virtual World Construction", October 1991.

[ZYDA 92] Zyda, Michael J, Pratt David R, Monahan Gregory, and Wilson, Kalin P; "NPSNET: Constructing a 3D Virtual World", *Symposium on 3D Graphics, '92 Proceedings*, April 1992, pp 147-156.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Dr. Michael J. Zyda Code CS/Zk Naval Postgraduate School Monterey, CA 93943-5000	6
4.	David R. Pratt Code CS/Pr Naval Postgraduate School Monterey, CA 93943-5000	6
5.	CPT Jon C. Walter P. O. Box 246 Hoxie, AR 72433	1
6.	CPT Patrick T. Warren 5000 Alabama St. #36 El Paso, TX 79930	1
7.	Director U.S. Army, TRADOC Analysis Command-Monterey Naval Postgraduate School Monterey, CA 93943	6
8.	Director U.S. Army TRADOC Analysis Command-WSMR ATTN: ATRC-WE White Sands Missle Range, NM 88602-5502	1
9.	Dr. Ralph M. Toms Conflict Simulation Laboratory Lawrence Livermore National Laboratory University of Califonia Livermore, CA 94551	1
10.	W.M. Christenson Strategy, Forces, and Resources Division Institute for Defense Analyses 1801 N. Beauregard Street Alexandria, VA 22311-1772	1